DEVOXX FRANCE

The Lightning Memory-Mapped Database (LMDB)

14:35h - 15:25h - Salle Ella Fitzgerald

The Lightning Memory-Mapped Database (LMDB)

Howard Chu
CTO, Symas Corp.
Chief Architect, OpenLDAP

@hyc_symas

Howard Chu

- Founder and CTO Symas Corp.
  - Member OpenLDAP since 1999, Chief Architect 2007
- Developing Free/Open Source software since 1980s
  - GNU compiler toolchain, e.g. "gmake -j", etc.
  - Too many projects to list...
- Worked for NASA/JPL, wrote software for Space Shuttle

Topics

Background

Features

API Overview

Internals

Special Features

Results

Background

API Inspired by Berkeley DB (BDB)

- OpenLDAP has used BDB extensively since 1999
- Deep experience with pros and cons of BDB design and implementation
- Omits BDB features that were found to be of no benefit
  - e.g. extensible hashing
- Avoids BDB characteristics that were problematic
  - e.g. cache tuning, complex locking, transaction logs, recovery

Features

OpenLDAP LMDB At A Glance

- Key/Value store using B+trees
- Fully transactional, ACID compliant
- MVCC, readers never block
- Uses memory-mapped files, needs no tuning
- Crash-proof, no recovery needed after restart
- Highly optimized, extremely compact
  - under 40KB object code, fits in CPU L1 Icache
- Runs on most modern OSs
  - Linux, Android, *BSD, MacOSX, Solaris, Windows, etc...

Features

Concurrency Support

- Both multi-process and multi-thread
- Single Writer + N Readers
  - Writers don't block readers
  - Readers don't block writers
  - Reads scale perfectly linearly with available CPUs
  - No deadlocks
- Full isolation with MVCC
- Nested transactions
- Batched writes

Features

Uses Copy-on-Write

- Live data is never overwritten
- Database structure cannot be corrupted by incomplete operations (system crashes)
- No write-ahead logs needed
- No transaction log cleanup/maintenance
- No recovery needed after crashes

Features

Uses Single-Level-Store

- Reads are satisfied directly from the memory map
  - no malloc or memcpy overhead
- Writes can be performed directly to the memory map
  - no write buffers, no buffer tuning
- Relies on the OS/filesystem cache
  - no wasted memory in app-level caching
- Can store live pointer-based objects directly
  - using a fixed address map
  - minimal marshalling, no unmarshalling required

API Overview

- Based on BDB Transactional API
  - BDB apps can easily be migrated to LMDB
- Written in C
  - C/C++ supported directly
  - Wrappers for Erlang, Lua, Python, Ruby available
  - Wrappers for other languages coming, as-needed
- All functions return 0 on success or a non-zero error code on failure
  - except some void functions which cannot fail

API Overview

- All DB operations are transactional
  - There is no non-transactional interface
- Results fetched from the DB are owned by the DB
  - Point directly to the mmap contents, not memcpy'd
  - Need no disposal, callers can use the data then forget about it
  - Read-only by default, attempts to overwrite data will trigger a SIGSEGV

API Overview

Most function names are grouped by purpose:

- Environment:
  - mdb_env_create, mdb_env_open, mdb_env_sync, mdb_env_close
- Transaction:
  - mdb_txn_begin, mdb_txn_commit, mdb_txn_abort
- Cursor:
  - mdb_cursor_open, mdb_cursor_close, mdb_cursor_get, mdb_cursor_put, mdb_cursor_del
- Database/Generic:
  - mdb_open, mdb_close, mdb_get, mdb_put, mdb_del

API Overview

LMDB Sample

```
#include <stdio.h>
#include <lmdb.h>

int main(int argc, char *argv[])
{
    int rc;
    MDB_env *env;
    MDB_txn *txn;
    MDB_cursor *cursor;
    MDB_dbi dbi;
    MDB_val key, data;
    char sval[32];

    rc = mdb_env_create(&env);
    rc = mdb_env_open(env,
      "./testdb", 0, 0664);
    rc = mdb_txn_begin(env, NULL,
      0, &txn);
    rc = mdb_open(txn, NULL, 0,
      &dbi);

    key.mv_size = sizeof(int);
    key.mv_data = sval;
    data.mv_size = sizeof(sval);
    data.mv_data = sval;
    sprintf(sval, "%03x %d foo bar", 32, 3141592);
    rc = mdb_put(txn, dbi, &key, &data, 0);
    rc = mdb_txn_commit(txn);
    if (rc) {
        fprintf(stderr, "mdb_txn_commit: (%d) %s\n",
          rc, mdb_strerror(rc));
        goto leave;
    }
    rc = mdb_txn_begin(env, NULL, MDB_RDONLY, &txn);
    rc = mdb_cursor_open(txn, dbi, &cursor);
    while ((rc = mdb_cursor_get(cursor, &key, &data,
      MDB_NEXT)) == 0) {
        printf("key: %p %.*s, data: %p %.*s\n",
            key.mv_data,
            (int) key.mv_size,
            (char *) key.mv_data,
            data.mv_data,
            (int) data.mv_size,
            (char *) data.mv_data);
    }
    mdb_cursor_close(cursor);
    mdb_txn_abort(txn);
leave:
    mdb_close(env, dbi);
    mdb_env_close(env);
    return rc;
}
```

API Overview

BDB Sample

```
#include <stdio.h>
#include <string.h>
#include <db.h>

int main(int argc, char *argv[])
{
    int rc;
    DB_ENV *env;
    DB_TXN *txn;
    DBC *cursor;
    DB *dbi;
    DBT key, data;
    char sval[32], kval[32];

#define FLAGS (DB_INIT_LOCK|DB_INIT_LOG|DB_INIT_TXN|
 DB_INIT_MPOOL|DB_CREATE|DB_THREAD)
    rc = db_env_create(&env, 0);
    rc = env->open(env, "./testdb", FLAGS,
      0664);
    rc = db_create(&dbi, env, 0);
    rc = env->txn_begin(env, NULL, &txn, 0);
    rc = dbi->open(dbi, txn, "test.bdb", NULL,
      DB_BTREE, DB_CREATE, 0664);

    memset(&key, 0, sizeof(DBT));
    memset(&data, 0, sizeof(DBT));
    key.size = sizeof(int);
    key.data = sval;
    data.size = sizeof(sval);
    data.data = sval;
    sprintf(sval, "%03x %d foo bar", 32, 3141592);
    rc = dbi->put(dbi, txn, &key, &data, 0);
    rc = txn->commit(txn, 0);
    if (rc) {
        fprintf(stderr, "txn->commit: (%d) %s\n",
          rc, db_strerror(rc));
        goto leave;
    }
    rc = env->txn_begin(env, NULL, &txn, 0);
    rc = dbi->cursor(dbi, txn, &cursor, 0);
    key.flags = DB_DBT_USERMEM;
    key.data = kval;
    key.ulen = sizeof(kval);
    data.flags = DB_DBT_USERMEM;
    data.data = sval;
    data.ulen = sizeof(sval);
    while ((rc = cursor->c_get(cursor, &key, &data,
      DB_NEXT)) == 0) {
        printf("key: %p %.*s, data: %p %.*s\n",
            key.data,
            (int) key.size,
            (char *) key.data,
            data.data,
            (int) data.size,
            (char *) data.data);
    }
    rc = cursor->c_close(cursor);
    rc = txn->abort(txn);
leave:
    rc = dbi->close(dbi, 0);
    rc = env->close(env, 0);
    return rc;
}
```

API Overview

LMDB Sample

```
#include <stdio.h>

#include <lmdb.h>

int main(int argc, char *argv[])
{
    int rc;
    MDB_env *env;
    MDB_txn *txn;
    MDB_cursor *cursor;
    MDB_dbi dbi;
    MDB_val key, data;
    char sval[32];




    rc = mdb_env_create(&env);
    rc = mdb_env_open(env, "./testdb", 0,
      0664);

    rc = mdb_txn_begin(env, NULL, 0, &txn);
    rc = mdb_open(txn, NULL, 0, &dbi);
```

BDB Sample

```
#include <stdio.h>
#include <string.h>
#include <db.h>

int main(int argc, char *argv[])
{
    int rc;
    DB_ENV *env;
    DB_TXN *txn;
    DBC *cursor;
    DB *dbi;
    DBT key, data;
    char sval[32], kval[32];

#define FLAGS (DB_INIT_LOCK|DB_INIT_LOG|DB_INIT_TXN|DB_INIT_MPOOL|
   DB_CREATE|DB_THREAD)
    rc = db_env_create(&env, 0);
    rc = env->open(env, "./testdb", FLAGS,
      0664);
    rc = db_create(&dbi, env, 0);
    rc = env->txn_begin(env, NULL, &txn, 0);
    rc = dbi->open(dbi, txn, "test.bdb",
      NULL, DB_BTREE, DB_CREATE, 0664);
```

API Overview

LMDB Sample

```
 key.mv_size = sizeof(int);
 key.mv_data = sval;
 data.mv_size = sizeof(sval);
 data.mv_data = sval;
 sprintf(sval, "%03x %d foo bar",
   32, 3141592);
 rc = mdb_put(txn, dbi, &key, &data, 0);
 rc = mdb_txn_commit(txn);
 if (rc) {
     fprintf(stderr, "mdb_txn_commit: (%d) %s\n", rc,
mdb_strerror(rc));
     goto leave;
 }
```

BDB Sample

```
memset(&key, 0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));
key.size = sizeof(int);
key.data = sval;
data.size = sizeof(sval);
data.data = sval;
sprintf(sval, "%03x %d foo bar",
  32, 3141592);
rc = dbi->put(dbi, txn, &key, &data, 0);
rc = txn->commit(txn, 0);
if (rc) {
    fprintf(stderr, "txn->commit: (%d) %s\n", rc, db_strerror(rc));
    goto leave;
}
```

API Overview

LMDB Sample

```
    rc = mdb_txn_begin(env, NULL, MDB_RDONLY,
      &txn);
    rc = mdb_cursor_open(txn, dbi, &cursor);







    while ((rc = mdb_cursor_get(cursor, &key,
      &data, MDB_NEXT)) == 0) {
       printf("key: %p %.*s, data: %p %.*s\n",
         key.mv_data,
         (int)key.mv_size,
         (char *)key.mv_data,
         data.mv_data,
         (int)data.mv_size,
         (char *)data.mv_data);
    }
    mdb_cursor_close(cursor);
    mdb_txn_abort(txn);
leave:
    mdb_close(env, dbi);
    mdb_env_close(env);
    return rc;
}
```

BDB Sample

```
    rc = env->txn_begin(env, NULL, &txn,
      0);
    rc = dbi->cursor(dbi, txn, &cursor, 0);
    key.flags = DB_DBT_USERMEM;
    key.data = kval;
    key.ulen = sizeof(kval);
    data.flags = DB_DBT_USERMEM;
    data.data = sval;
    data.ulen = sizeof(sval);
    while ((rc = cursor->c_get(cursor, &key,
      &data, DB_NEXT)) == 0) {
        printf("key: %p %.*s, data: %p %.*s\n",
          key.data,
          (int)key.size,
          (char *)key.data,
          data.data,
          (int)data.size,
          (char *)data.data);
    }
    rc = cursor->c_close(cursor);
    rc = txn->abort(txn);
leave:
    rc = dbi->close(dbi, 0);
    rc = env->close(env, 0);
    return rc;
}
```

API Overview

- LMDB naming is simple and consistent
  - MDB_xxx for all typedefs
  - BDB uses DB_XXX, DBX, DB_DBX_...
- LMDB environment setup is simple
  - BDB requires multiple subsystems to be initialized
- LMDB database setup is simple and reliable
  - BDB creates a file per DB
    - If the transaction containing the DB Open is aborted, rollback is very complicated because the filesystem operations to create the file cannot be rolled back atomically
    - Likewise during recovery and replay of a transaction log

API Overview

- LMDB data is simple
  - BDB requires DBT structure to be fully zeroed out before use
  - BDB requires the app to manage the memory of keys and values returned from the DB
- LMDB teardown is simple
  - BDB *-close functions can fail, and there's nothing the app can do if a failure occurs

API Overview

LMDB config is simple, e.g. slapd

```
database mdb
directory /var/lib/ldap/data/mdb
maxsize 4294967296
```

BDB config is complex

```
database hdb
directory /var/lib/ldap/data/hdb
cachesize 50000
idlcachesize 50000
dbconfig set_cachesize 4 0 1
dbconfig set_lg_regionmax 262144
dbconfig set_lg_bsize 2097152
dbconfig set_lg_dir /mnt/logs/hdb
dbconfig set_lk_max_locks 3000
dbconfig set_lk_max_objects 1500
dbconfig set_lk_max_lockers 1500
```

Internals

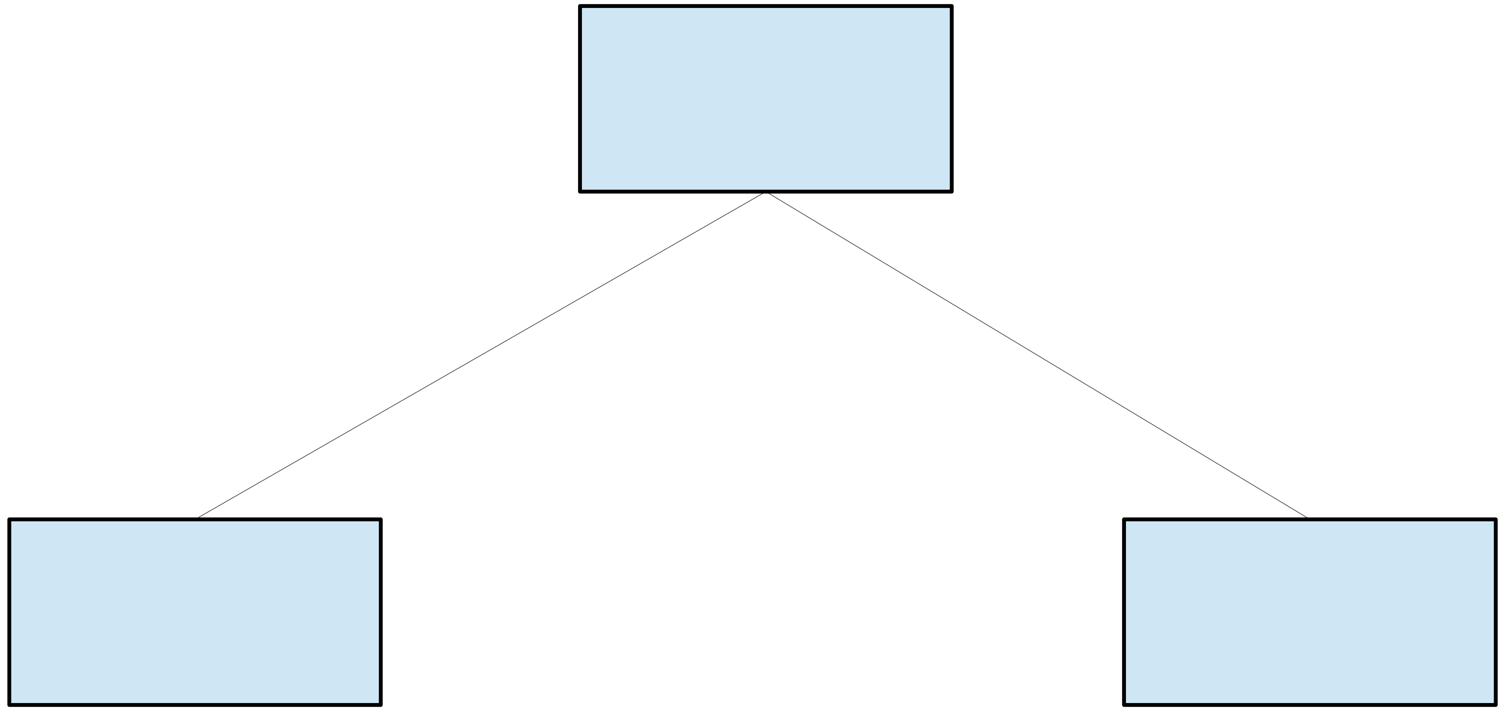
- B+tree Operation
  - Append-only, Copy-on-Write
  - Corruption-Proof
- Free Space Management
  - Avoiding Compaction/Garbage Collection
- Transaction Handling
  - Avoiding Locking

B+tree Operation

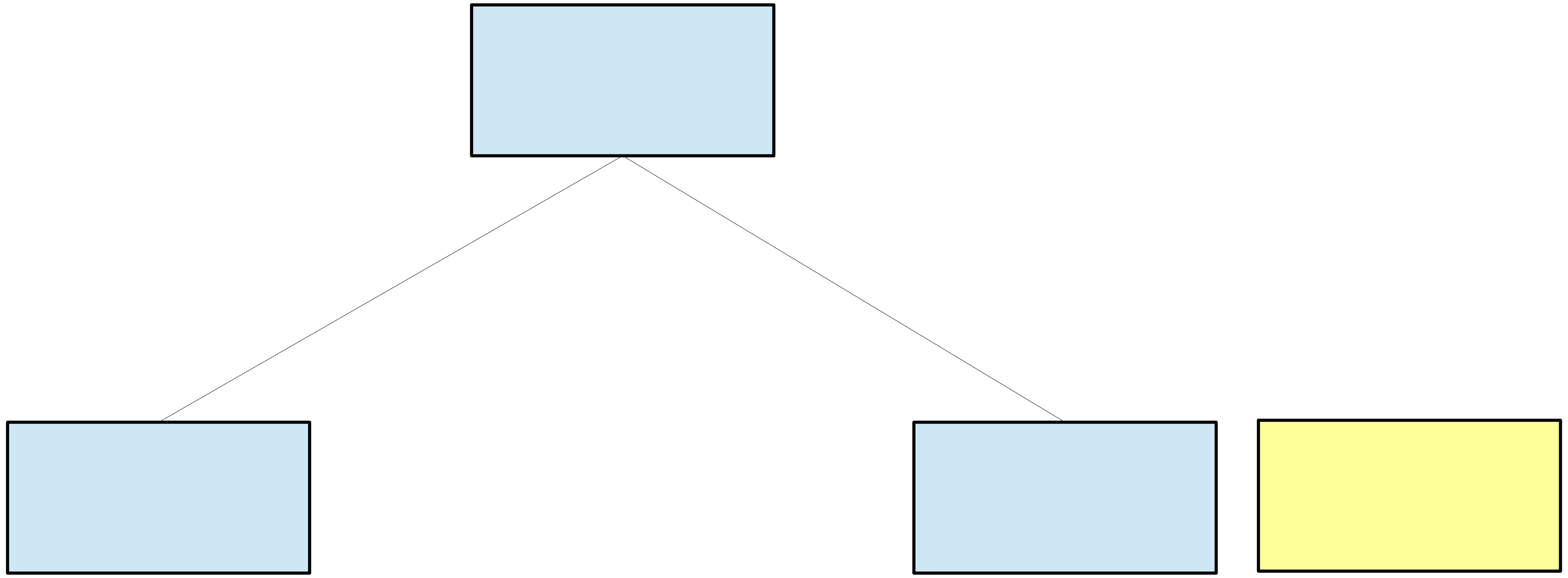
How Append-Only/Copy-On-Write Works

- In a pure append-only approach, no data is ever overwritten
- Pages that are meant to be modified are copied
- The modification is made on the copy
- The copy is written to a new disk page
- The structure is inherently multi-version; you can find any previous version of the database by starting at the root node corresponding to that version

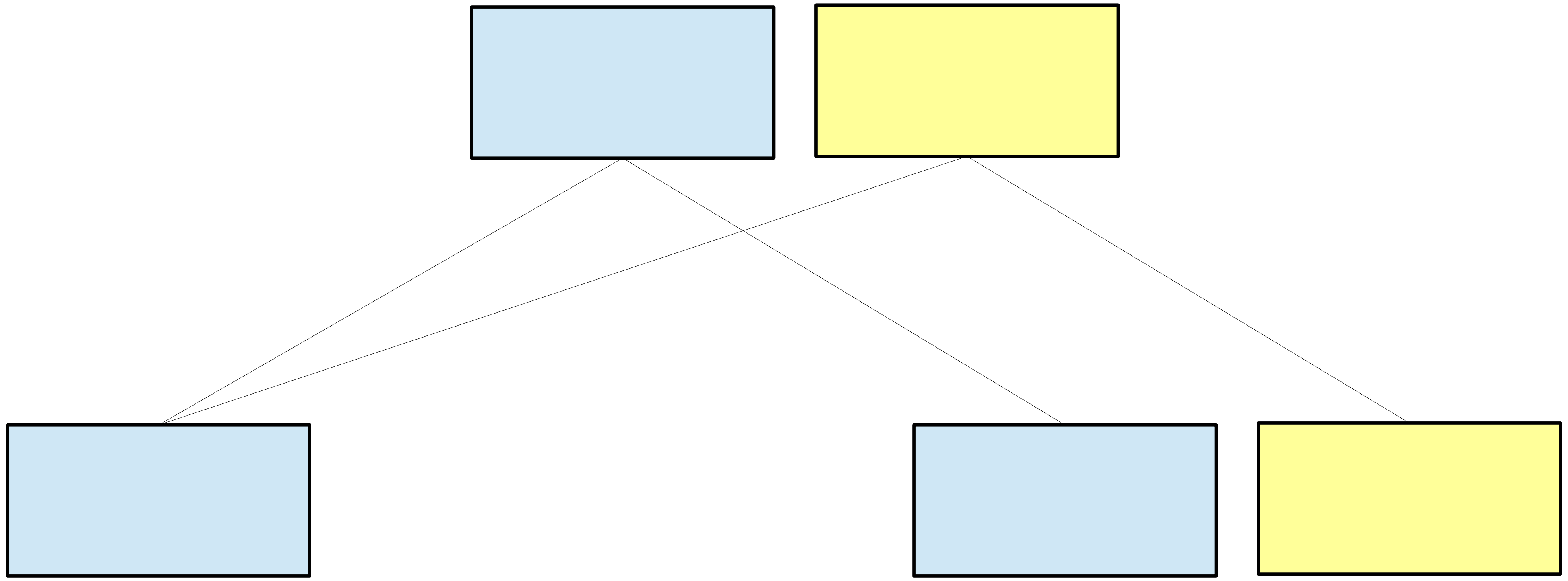
B+tree Operation



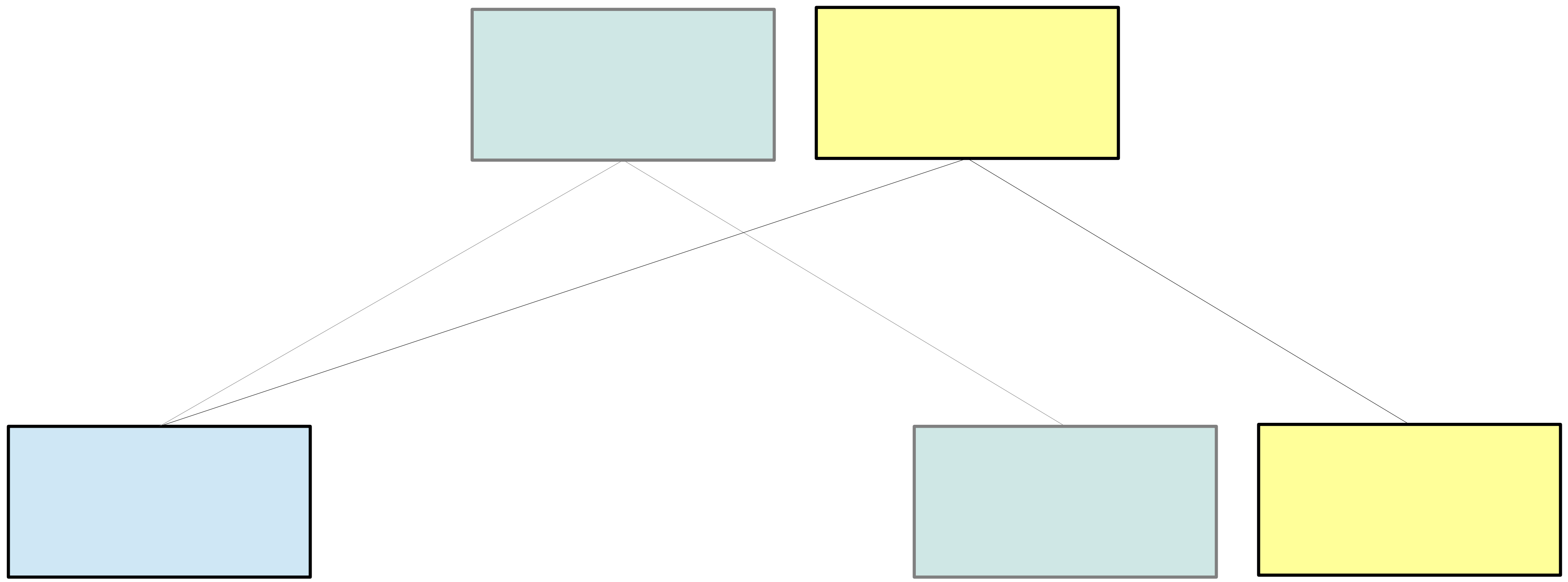
B+tree Operation



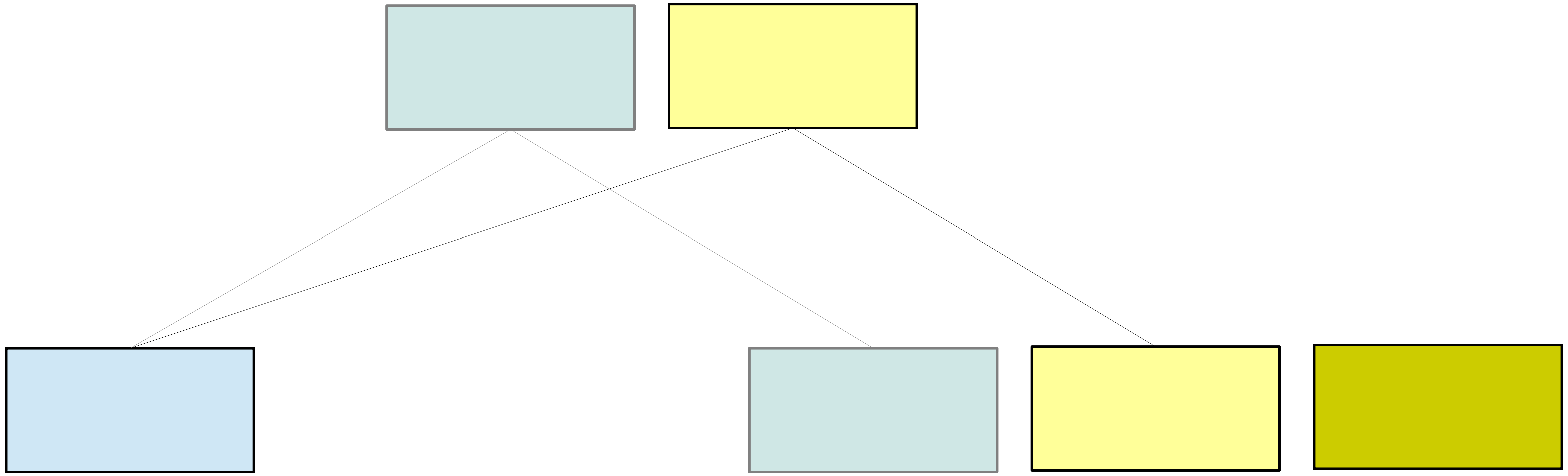
B+tree Operation



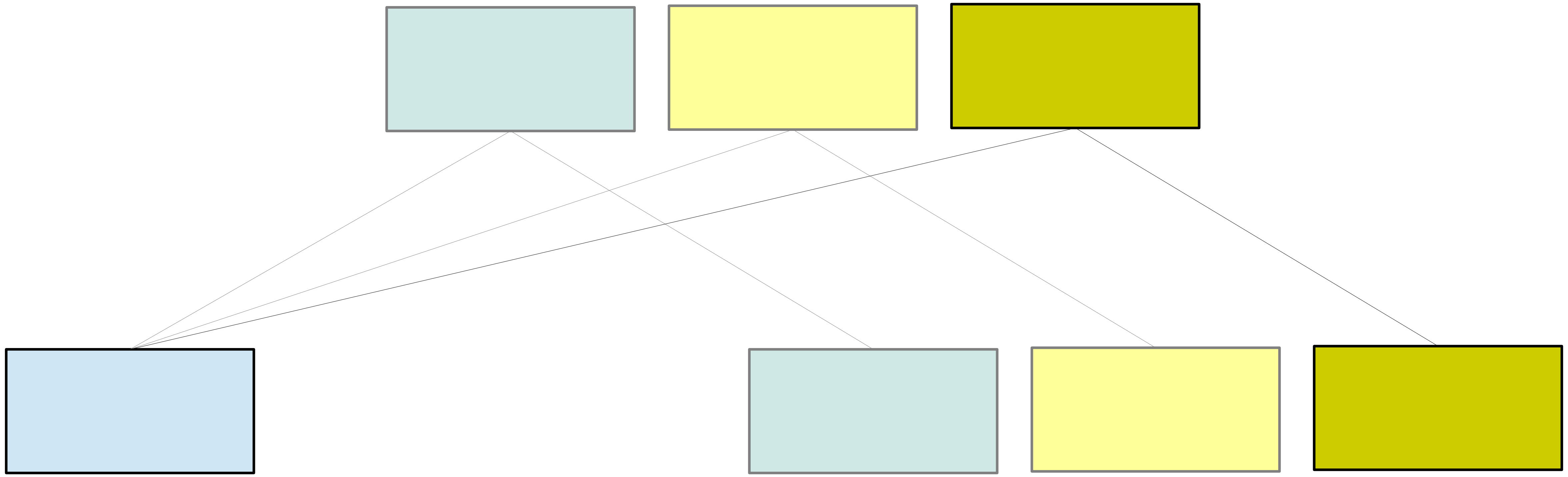
B+tree Operation



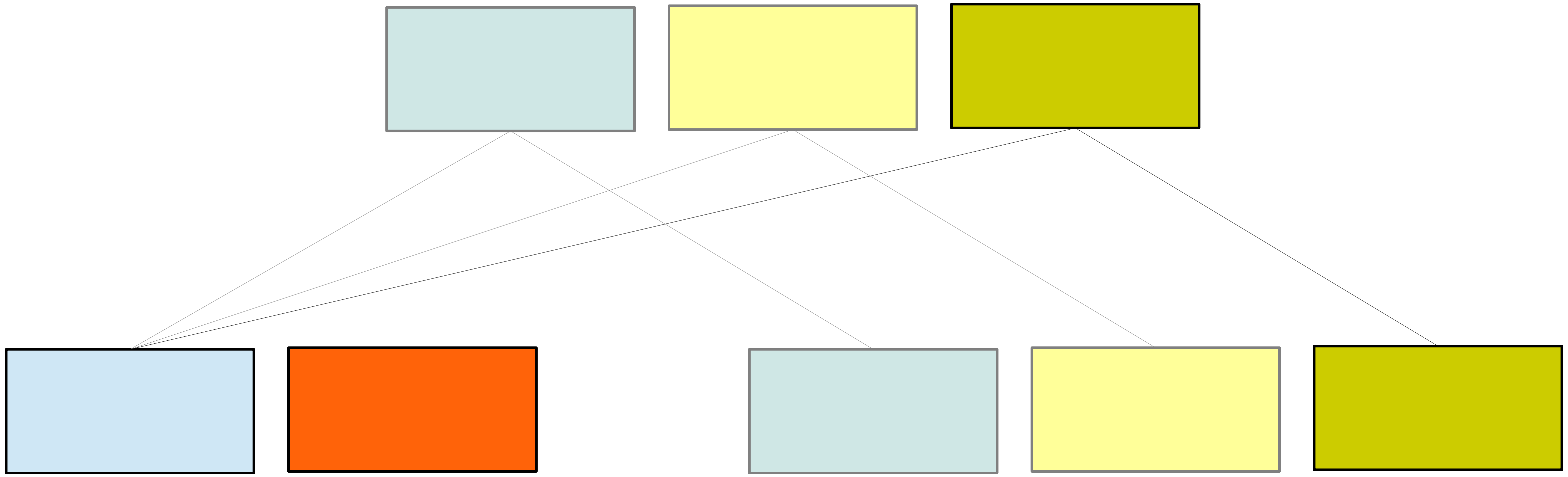
B+tree Operation



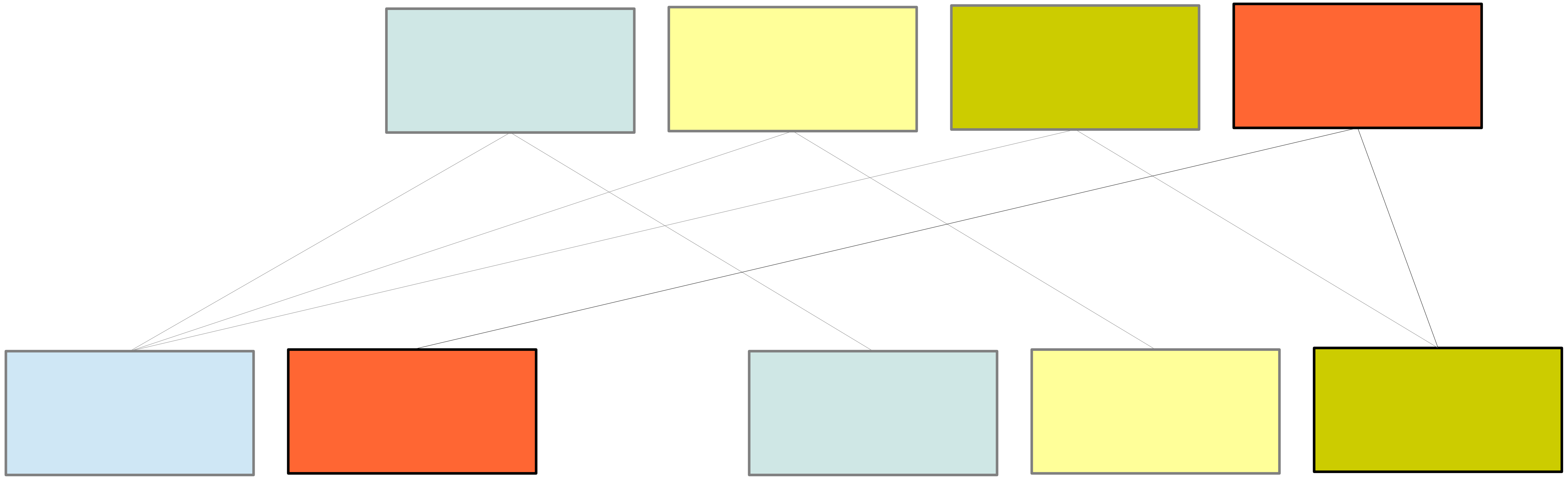
B+tree Operation



B+tree Operation



B+tree Operation



B+tree Operation

How Append-Only/Copy-On-Write Works

- Updates are always performed bottom up
- Every branch node from the leaf to the root must be copied/modified for any leaf update
- Any node not on the path from the leaf to the root is left unaltered
- The root node is always written last

B+tree Operation

In the Append-Only tree, new pages are always appended sequentially to the database file

- While there's significant overhead for making complete copies of modified pages, the actual I/O is linear and relatively fast
- The root node is always the last page of the file, unless there was a system crash
- Any root node can be found by searching backward from the end of the file, and checking the page's header
- Recovery from a system crash is relatively easy
  - Everything from the last valid root to the beginning of the file is always pristine
  - Anything between the end of the file and the last valid root is discarded

B+tree Operation

Append-Only disk usage is very inefficient

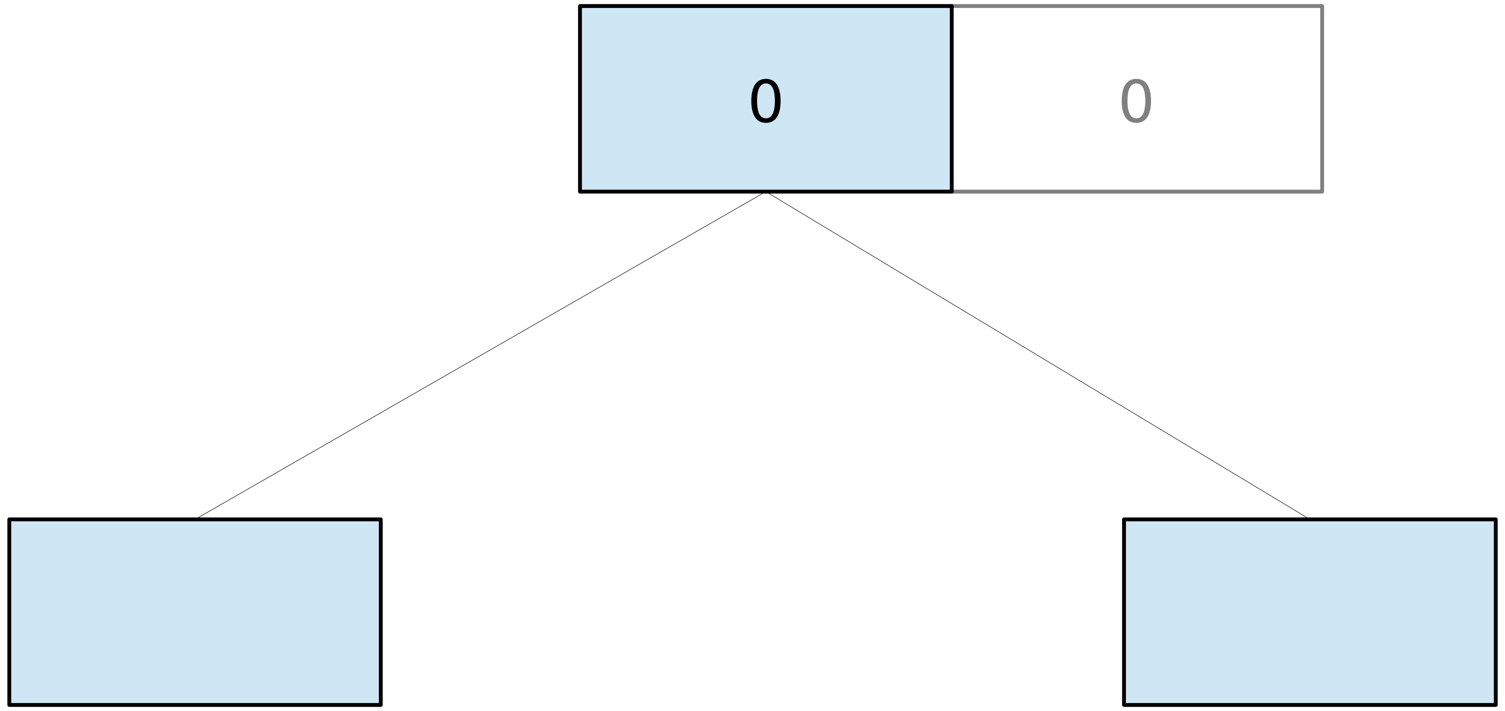
- Disk space usage grows without bound
- 99+% of the space will be occupied by old versions of the data
- The old versions are usually not interesting
- Reclaiming the old space requires a very expensive compaction phase
- New updates must be throttled until compaction completes

B+tree Operation

The LMDB Approach

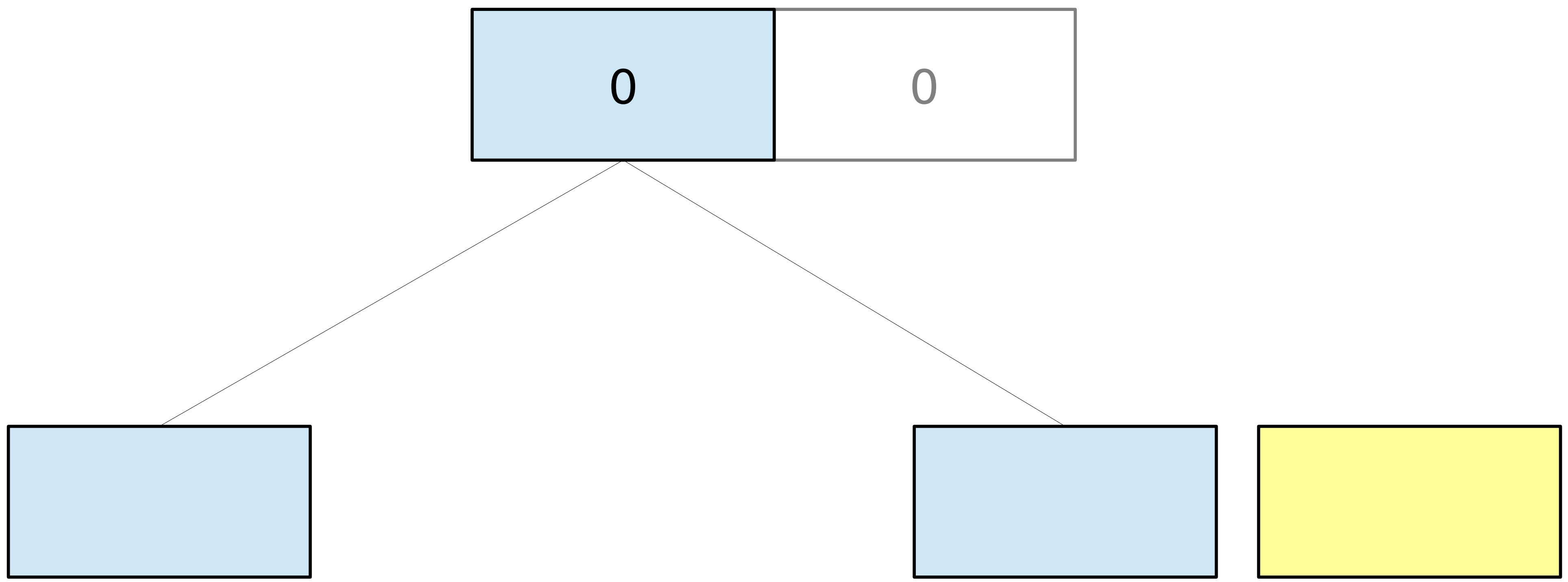
- Still Copy-on-Write, but using two fixed root nodes
  - Page 0 and Page 1 of the file, used in double-buffer fashion
  - Even faster cold-start than Append-Only, no searching needed to find the last valid root node
  - Any app always reads both pages and uses the one with the greater Transaction ID stamp in its header
  - Consequently, only 2 outstanding versions of the DB exist, not fully "multi-version"

B+tree Operation

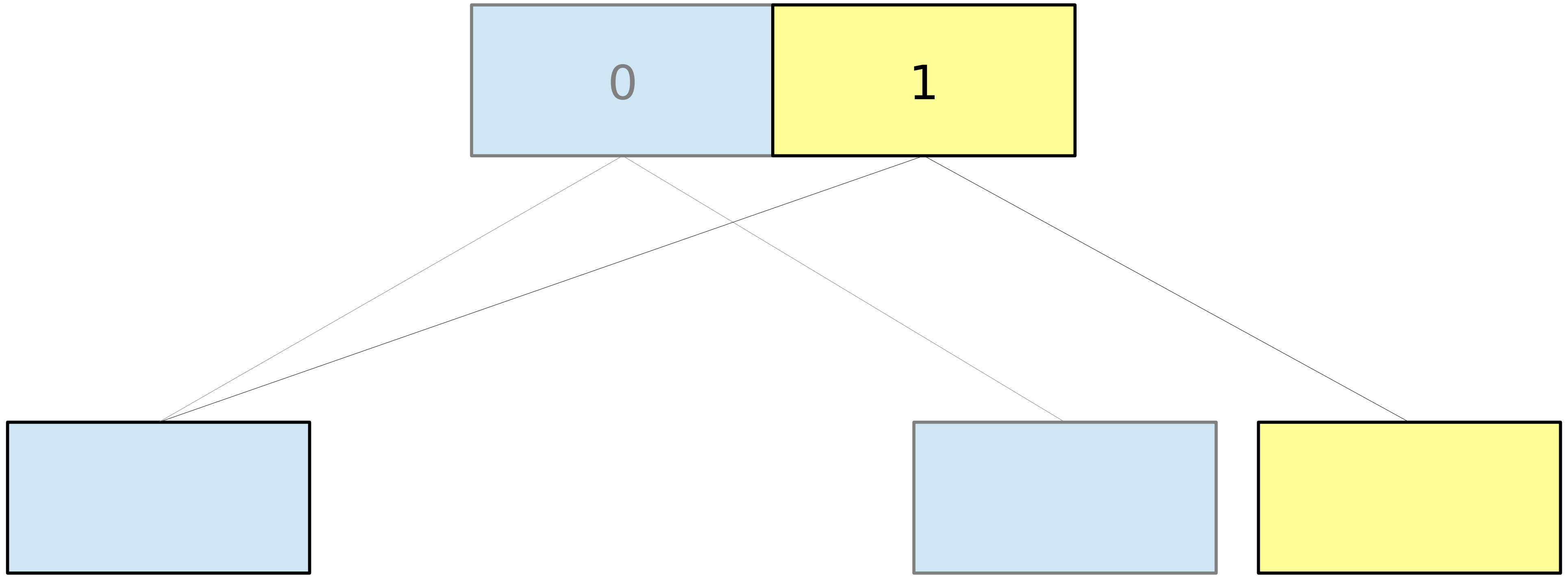


The root nodes have a transaction ID stamp

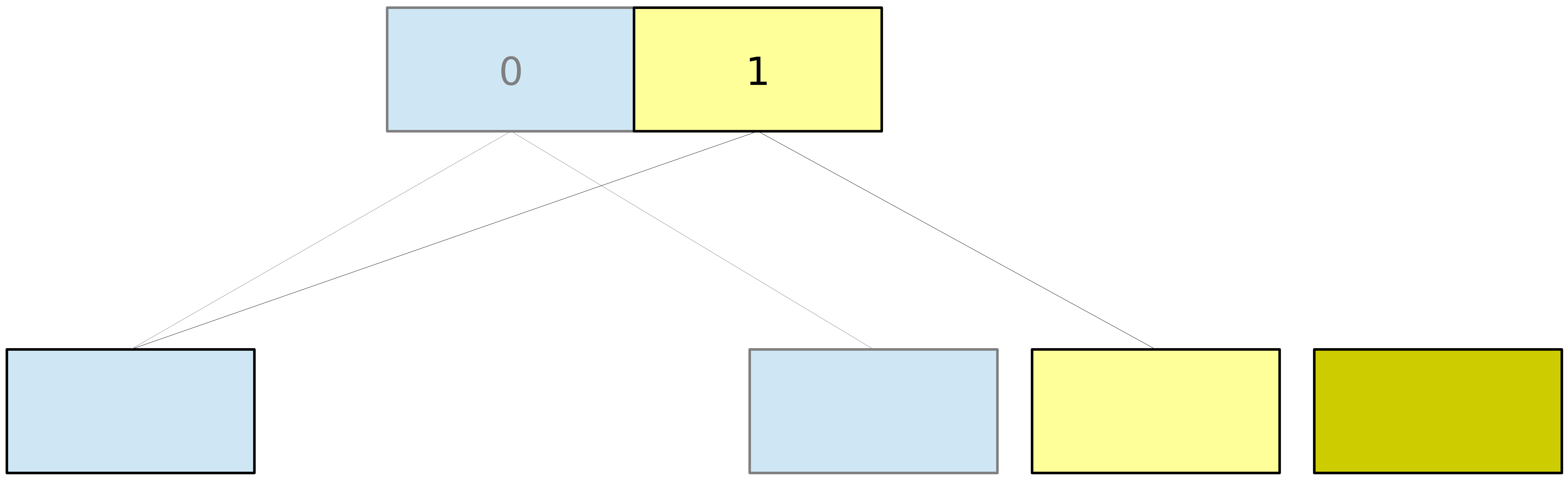
B+tree Operation



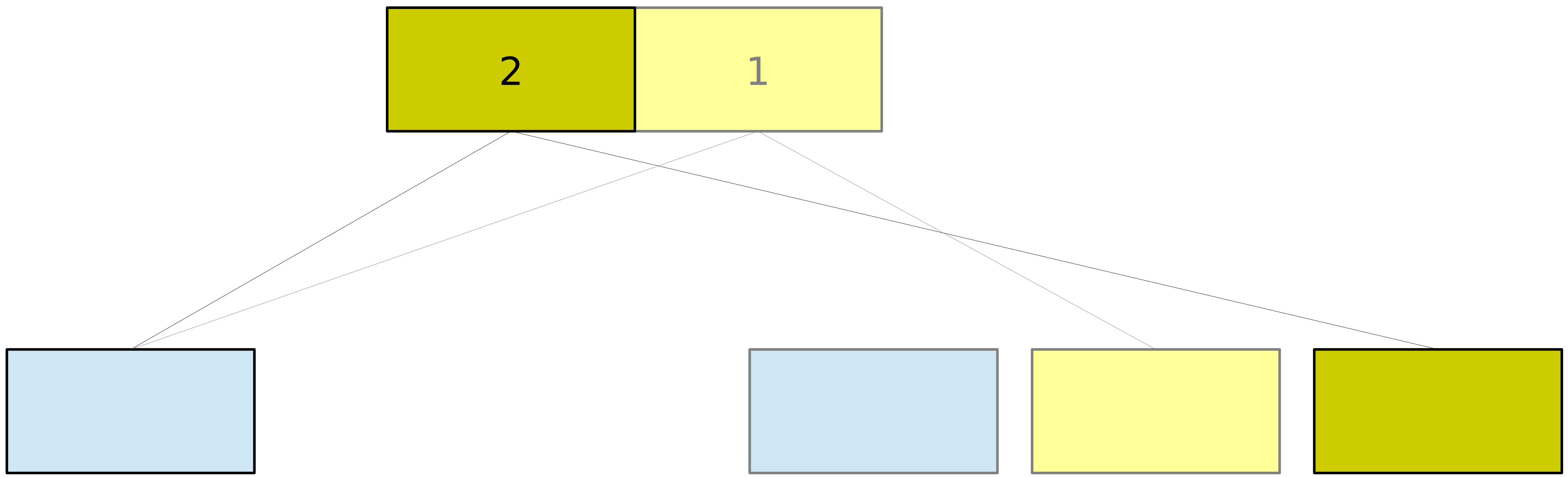
B+tree Operation



B+tree Operation

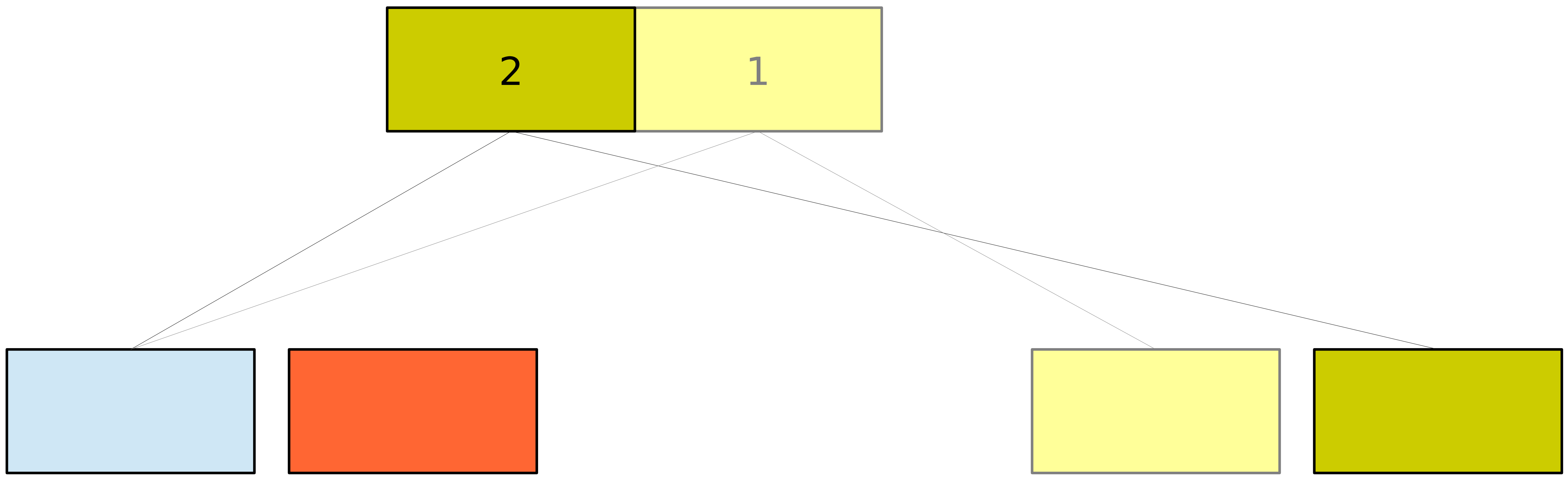


B+tree Operation

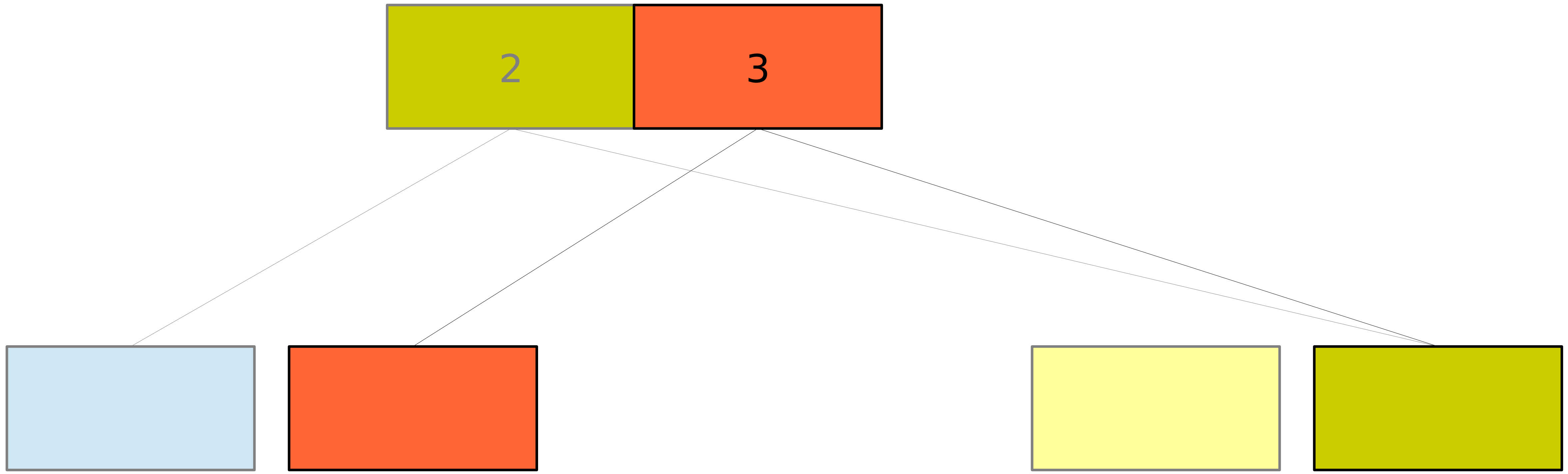


After this step the old blue page is no longer referenced by anything else in the database, so it can be reclaimed

B+tree Operation



B+tree Operation



After this step the old yellow page is no longer referenced by anything else in the database, so it can also be reclaimed

Free Space Management

LMDB maintains two B+trees per root node

- One storing the user data, as illustrated above
- One storing lists of IDs of pages that have been freed in a given transaction
- Old, freed pages are used in preference to new pages, so the DB file size remains relatively static over time
- No compaction or garbage collection phase is ever needed

Free Space Management

Pgno: 0 Misc... TXN: 0 FRoot: EMPTY DRoot: EMPTY	Pgno: 1 Misc... TXN: 0 FRoot: EMPTY DRoot: EMPTY

Free Space Management

Pgno: 0 Misc... TXN: 0 FRoot: EMPTY DRoot: EMPTY	Pgno: 1 Misc... TXN: 0 FRoot: EMPTY DRoot: EMPTY	Pgno: 2 Misc... offset: 4000 1,foo

Free Space Management

Pgno: 0 Misc... TXN: 0 FRoot: EMPTY DRoot: EMPTY	Pgno: 1 Misc... TXN: 1 FRoot: EMPTY DRoot: 2	Pgno: 2 Misc... offset: 4000 1,foo

Free Space Management

Pgno: 0 Misc... TXN: 0 FRoot: EMPTY DRoot: EMPTY	Pgno: 1 Misc... TXN: 1 FRoot: EMPTY DRoot: 2	Pgno: 2 Misc... offset: 4000 1,foo	Pgno: 3 Misc... offset: 4000 offset: 3000 2,bar 1,foo

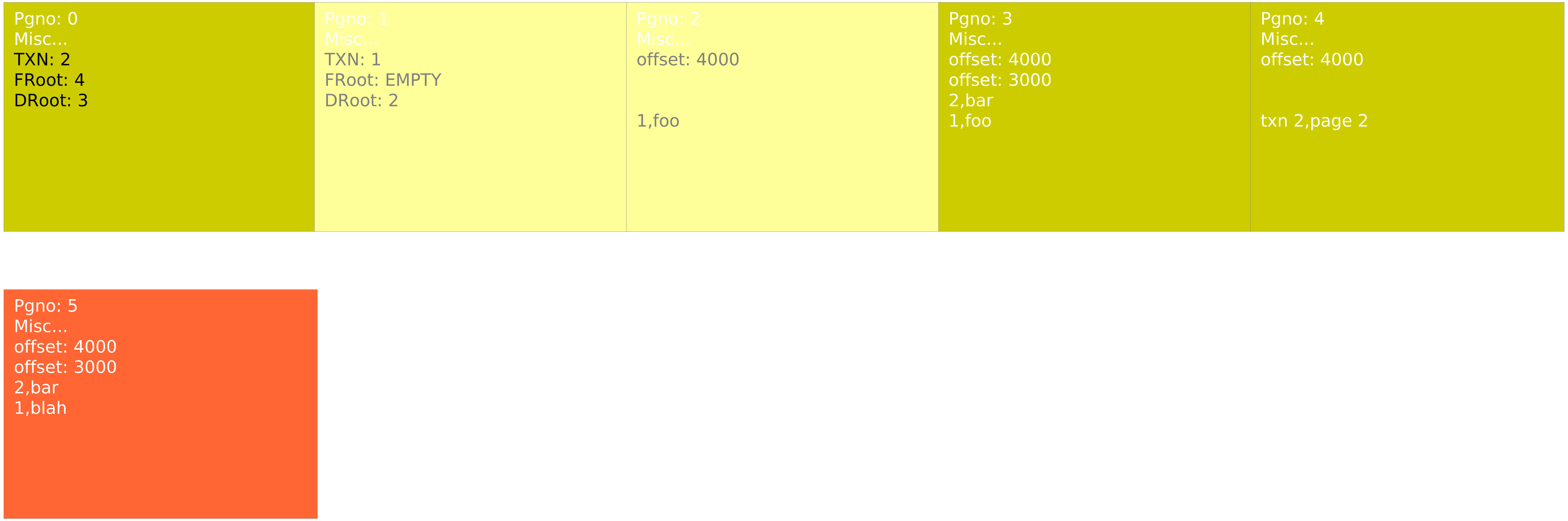
Free Space Management

Pgno: 0 Misc... TXN: 0 FRoot: EMPTY DRoot: EMPTY	Pgno: 1 Misc... TXN: 1 FRoot: EMPTY DRoot: 2	Pgno: 2 Misc... offset: 4000 1,foo	Pgno: 3 Misc... offset: 4000 offset: 3000 2,bar 1,foo	Pgno: 4 Misc... offset: 4000 txn 2,page 2

Free Space Management

Pgno: 0 Misc… TXN: 2 FRoot: 4 DRoot: 3	Pgno: 1 Misc… TXN: 1 FRoot: EMPTY DRoot: 2	Pgno: 2 Misc… offset: 4000 1,foo	Pgno: 3 Misc… offset: 4000 offset: 3000 2,bar 1,foo	Pgno: 4 Misc… offset: 4000 txn 2,page 2

Free Space Management



Free Space Management

Pgno: 0	Pgno: 1	Pgno: 2	Pgno: 3	Pgno: 4
Misc…	Misc…	Misc…	Misc…	Misc…
TXN: 2	TXN: 1	offset: 4000	offset: 4000	offset: 4000
FRoot: 4	FRoot: EMPTY		offset: 3000	
DRoot: 3	DRoot: 2		2,bar	
		1,foo	1,foo	txn 2,page 2

Pgno: 5	Pgno: 6
Misc…	Misc…
offset: 4000	offset: 4000
offset: 3000	offset: 3000
2,bar	txn 3,page 3,4
1,blah	txn 2,page 2

Free Space Management

Pgno: 0 Misc... TXN: 2 FRoot: 4 DRoot: 3	Pgno: 1 Misc... TXN: 3 FRoot: 6 DRoot: 5	Pgno: 2 Misc... offset: 4000 1,foo	Pgno: 3 Misc... offset: 4000 offset: 3000 2,bar 1,foo	Pgno: 4 Misc... offset: 4000 txn 2,page 2
Pgno: 5 Misc... offset: 4000 offset: 3000 2,bar 1,blah	Pgno: 6 Misc... offset: 4000 offset: 3000 txn 3,page 3,4 txn 2,page 2			

Free Space Management

Pgno: 0
Misc...
TXN: 2
FRoot: 4
DRoot: 3

Pgno: 1
Misc...
TXN: 3
FRoot: 6
DRoot: 5

Pgno: 2
Misc...
offset: 4000
offset: 3000
2,xyz
1,blah

Pgno: 3
Misc...
offset: 4000
offset: 3000
2,bar
1,foo

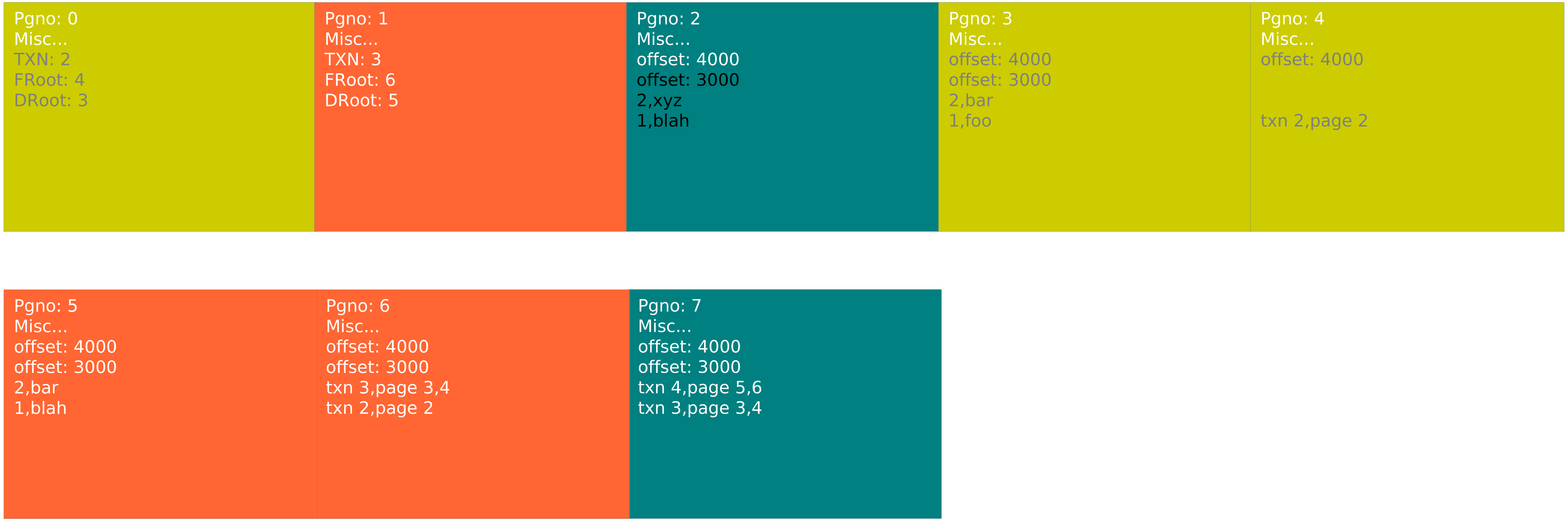
Pgno: 4
Misc...
offset: 4000

txn 2,page 2

Pgno: 5
Misc...
offset: 4000
offset: 3000
2,bar
1,blah

Pgno: 6
Misc...
offset: 4000
offset: 3000
txn 3,page 3,4
txn 2,page 2

Free Space Management



Free Space Management

Pgno: 0
Misc...
TXN: 4
FRoot: 7
DRoot: 2

Pgno: 1
Misc...
TXN: 3
FRoot: 6
DRoot: 5

Pgno: 2
Misc...
offset: 4000
offset: 3000
2,xyz
1,blah

Pgno: 3
Misc...
offset: 4000
offset: 3000
2,bar
1,foo

Pgno: 4
Misc...
offset: 4000

txn 2,page 2

Pgno: 5
Misc...
offset: 4000
offset: 3000
2,bar
1,blah

Pgno: 6
Misc...
offset: 4000
offset: 3000
txn 3,page 3,4
txn 2,page 2

Pgno: 7
Misc...
offset: 4000
offset: 3000
txn 4,page 5,6
txn 3,page 3,4

Free Space Management

- Caveat: If a read transaction is open on a particular version of the DB, that version and every version after it are excluded from page reclaiming
- Thus, long-lived read transactions should be avoided, otherwise the DB file size may grow rapidly, devolving into the Append-Only behavior until the transactions are closed

Transaction Handling

- LMDB supports a single writer concurrent with many readers
  - A single mutex serializes all write transactions
  - The mutex is shared/multiprocess
- Readers run lockless and never block
  - But for page reclamation purposes, readers are tracked
- Transactions are stamped with an ID which is a monotonically increasing integer
  - The ID is only incremented for Write transactions that actually modify data
  - If a Write transaction is aborted, or committed with no changes, the same ID will be reused for the next Write transaction

Transaction Handling

- Transactions take a snapshot of the currently valid meta page at the beginning of the transaction
- No matter what write transactions follow, a read transaction's snapshot will always point to a valid version of the DB
- The snapshot is totally isolated from subsequent writes
- This provides the Consistency and Isolation in ACID semantics

Transaction Handling

- The currently valid meta page is chosen based on the greatest transaction ID in each meta page
  - The meta pages are page and CPU cache aligned
  - The transaction ID is a single machine word
  - The update of the transaction ID is atomic
  - Thus, the Atomicity semantics of transactions are guaranteed

Transaction Handling

During Commit, the data pages are written and then synchronously flushed before the meta page is updated

- Then the meta page is written synchronously
- Thus, when a commit returns "success", it is guaranteed that the transaction has been written intact
- This provides the Durability semantics
- If the system crashes before the meta page is updated, then the data updates are irrelevant

Transaction Handling

For tracking purposes, Readers must acquire a slot in the readers table

- The readers table is also in a shared memory map, but separate from the main data map
- This is a simple array recording the Process ID, Thread ID, and Transaction ID of the reader
- The first time a thread opens a read transaction, it must acquire a mutex to reserve a slot in the table
- The slot ID is stored in Thread Local Storage; subsequent read transactions performed by the thread need no further locks

Transaction Handling

- Write transactions use pages from the free list before allocating new disk pages
  - Pages in the free list are used in order, oldest transaction first
  - The readers table must be scanned to see if any reader is referencing an old transaction
  - The writer doesn't need to lock the reader table when performing this scan - readers never block writers
    - The only consequence of scanning with no locks is that the writer may see stale data
    - This is irrelevant, newer readers are of no concern; only the oldest readers matter

Special Features

Explicit Key Types

- Support for reverse byte order comparisons, as well as native binary integer comparisons
- Minimizes the need for custom key comparison functions, allows DBs to be used safely by applications without special knowledge
  - Reduces the danger of corruption that Berkeley databases were vulnerable to, when custom key comparators were used

Special Features

Append Mode

- Ultra-fast writes when keys are added in sequential order
- Bypasses standard page-split algorithm when pages are filled, avoids unnecessary memcpy's
- Allows databases to be bulk loaded at the full sequential write speed of the underlying storage system

Special Features

Reserve Mode

- Allocates space in write buffer for data of user-specified size, returns address
- Useful for data that is generated dynamically instead of statically copied
- Allows generated data to be written directly to DB output buffer, avoiding unnecessary memcpy

Special Features

Fixed Mapping

- Uses a fixed address for the memory map
- Allows complex pointer-based data structures to be stored directly with minimal serialization
- Objects using persistent addresses can thus be read back with no deserialization
- Useful for object-oriented databases, among other purposes

Special Features

Sub-databases

- Store multiple independent named B+trees in a single LMDB environment
- A SubDB is simply a key/data pair in the main DB, where the data item is the root node of another tree
- Allows many related databases to be managed easily
  - Used in back-mdb for the main data and all of the associated indices
  - Used in SQLightning for multiple tables and indices

Special Features

Sorted Duplicates

- Allows multiple data values for a single key
- Values are stored in sorted order, with customizable comparison functions
- When the data values are all of a fixed size, the values are stored contiguously, with no extra headers
  - maximizes storage efficiency and performance
- Implemented by the same code as SubDB support
  - maximum coding efficiency

Special Features

Atomic Hot Backup

- The entire database can be backed up live
- No need to stop updates while backups run
- The backup runs at the maximum speed of the target storage medium
- Essentially: write(outfd, map, mapsize);
  - no memcpy's in or out of user space
  - pure DMA from the database to the backup

Results

Support for LMDB is already available for many open source projects:

- OpenLDAP slapd - back-mdb backend
- Cyrus SASL - sasldb plugin
- Heimdal Kerberos - hdb plugin
- OpenDKIM - main data store
- SQLite3 - replacing the original Btree code
- MemcacheDB - replacing BerkeleyDB
- Postfix - replacing BerkeleyDB
- CfEngine - replacing Tokyo Cabinet/QDBM

Results

Coming Soon

- Riak - Erlang LMDB wrapper already available
- SQLite4 - in progress
- MariaDB - in progress
- HyperDex - in progress
- XDAndroid - port of Android using SQLite3 based on LMDB
- Mozilla/Firefox - using SQLite3 based on LMDB

Results

- In OpenLDAP slapd
  - LMDB reads are 5-20x faster than BerkeleyDB
  - Writes are 2-5x faster than BerkeleyDB
  - Consumes 1/4 as much RAM as BerkeleyDB
- In SQLite3
  - Writes are 10-25x faster than stock SQLite3
  - Reads .. performance is overshadowed by SQL inefficiency

Results

In MemcacheDB

- LMDB reads are 2-200x faster than BerkeleyDB
- Writes are 5-900x faster than BerkeleyDB
- Multi-thread reads are 2-8x faster than pure-memory Memcached
  - Single-thread reads are about the same
  - Writes are about 20% slower

Results

- Full benchmark reports are available on the LMDB page
  - http://www.symas.com/mdb/
- Supported builds of LMDB-based packages available from Symas
  - http://www.symas.com/
  - OpenLDAP, Cyrus-SASL, Heimdal Kerberos
  - MemcacheDB coming soon

Microbenchmark Results

- Comparisons based on Google's LevelDB
- Also tested against Kyoto Cabinet's TreeDB, SQLite3, and BerkeleyDB
- Tested using RAM filesystem (tmpfs), reiserfs on SSD, and multiple filesystems on HDD
  - btrfs, ext2, ext3, ext4, jfs, ntfs, reiserfs, xfs, zfs
  - ext3, ext4, jfs, reiserfs, xfs also tested with external journals

Microbenchmark Results

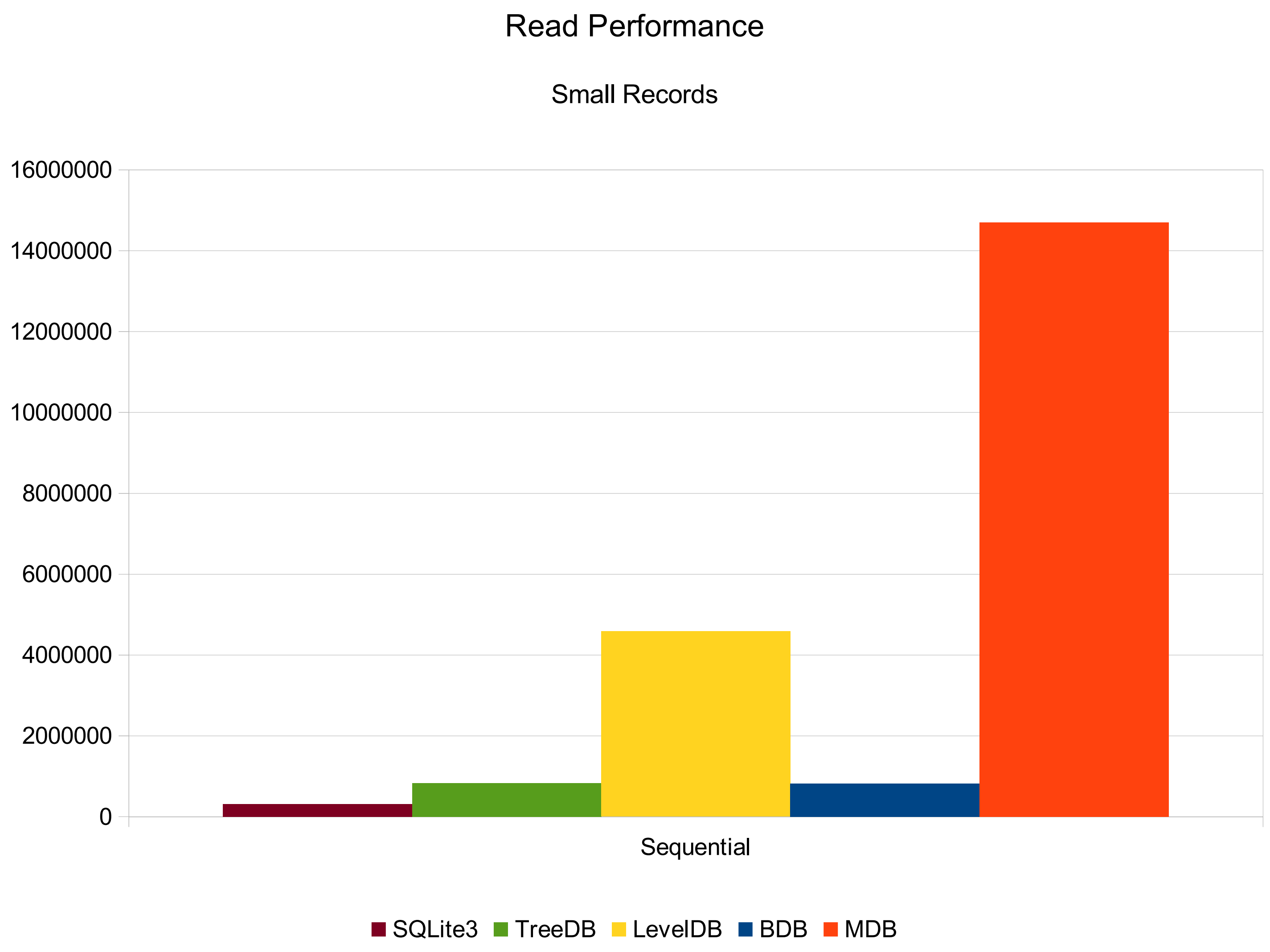
Relative Footprint

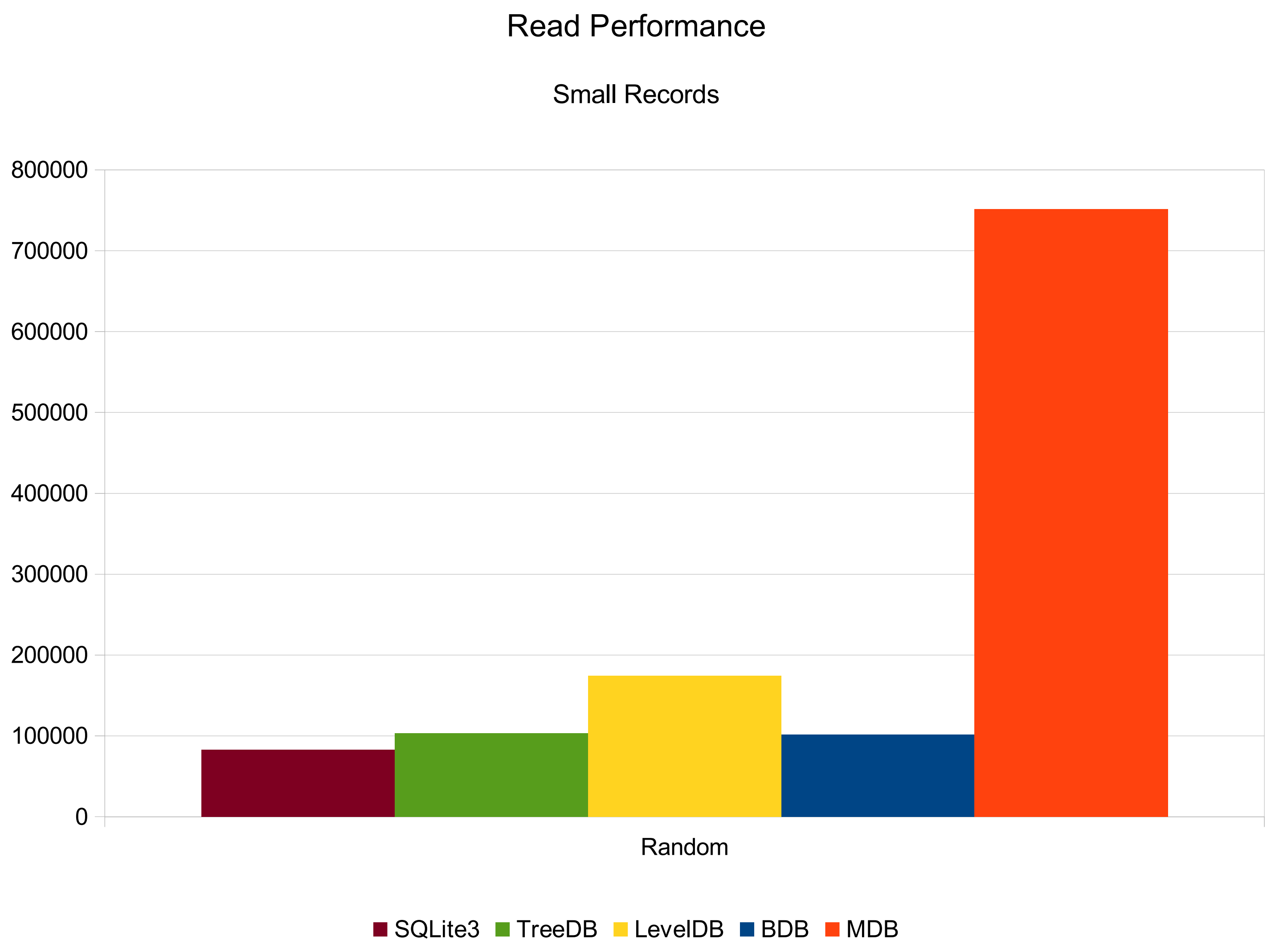
text	data	bss	dec	hex	filename
272247	1456	328	274031	42e6f	db_bench
1675911	2288	304	1678503	199ca7	db_bench_bdb
90423	1508	304	92235	1684b	db_bench_mdb
653480	7768	1688	662936	a2764	db_bench_sqlite3
296572	4808	1096	302476	49d8c	db_bench_tree_db

Clearly LMDB has the smallest footprint

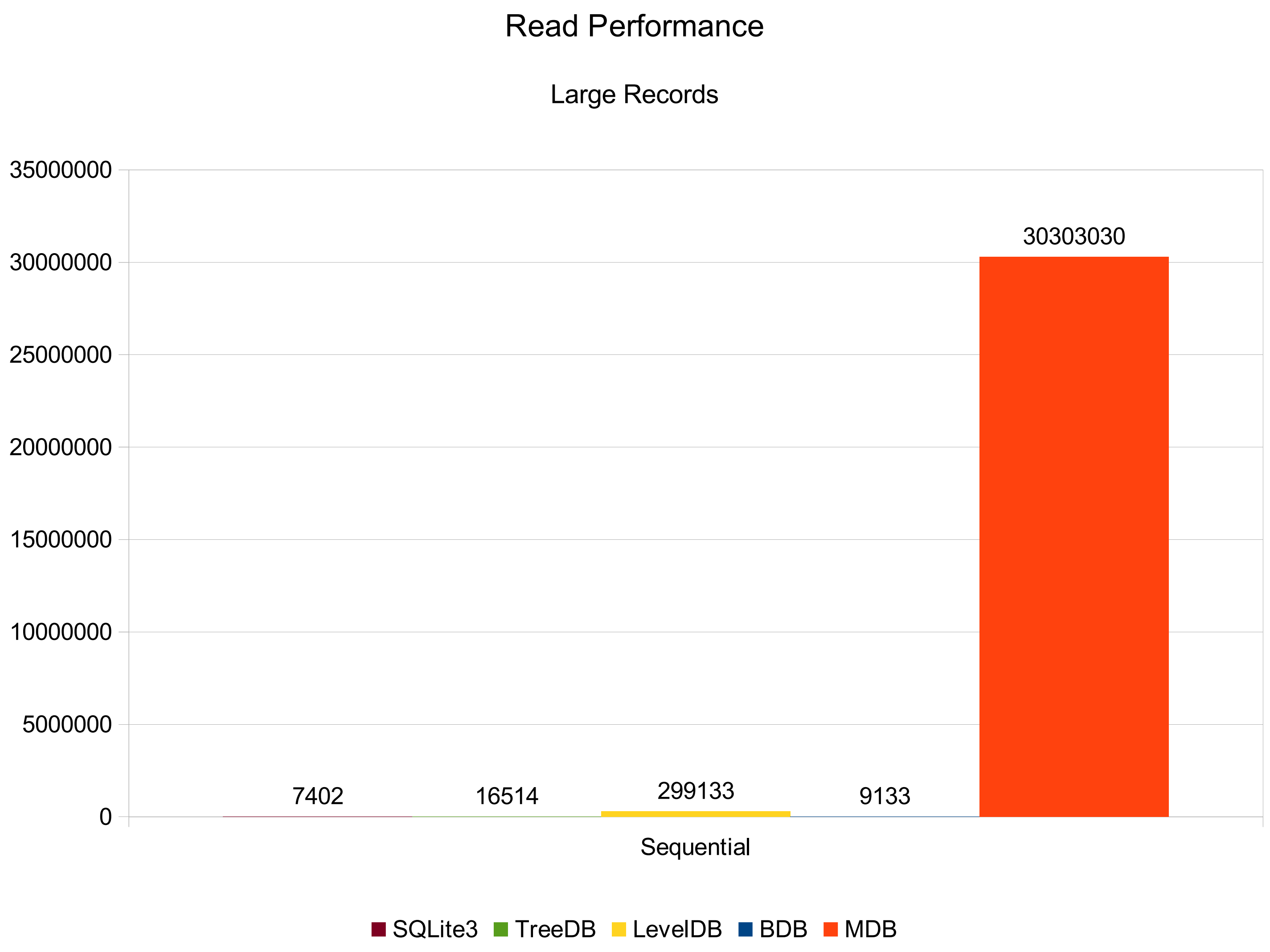
– Carefully written C code beats C++ every time

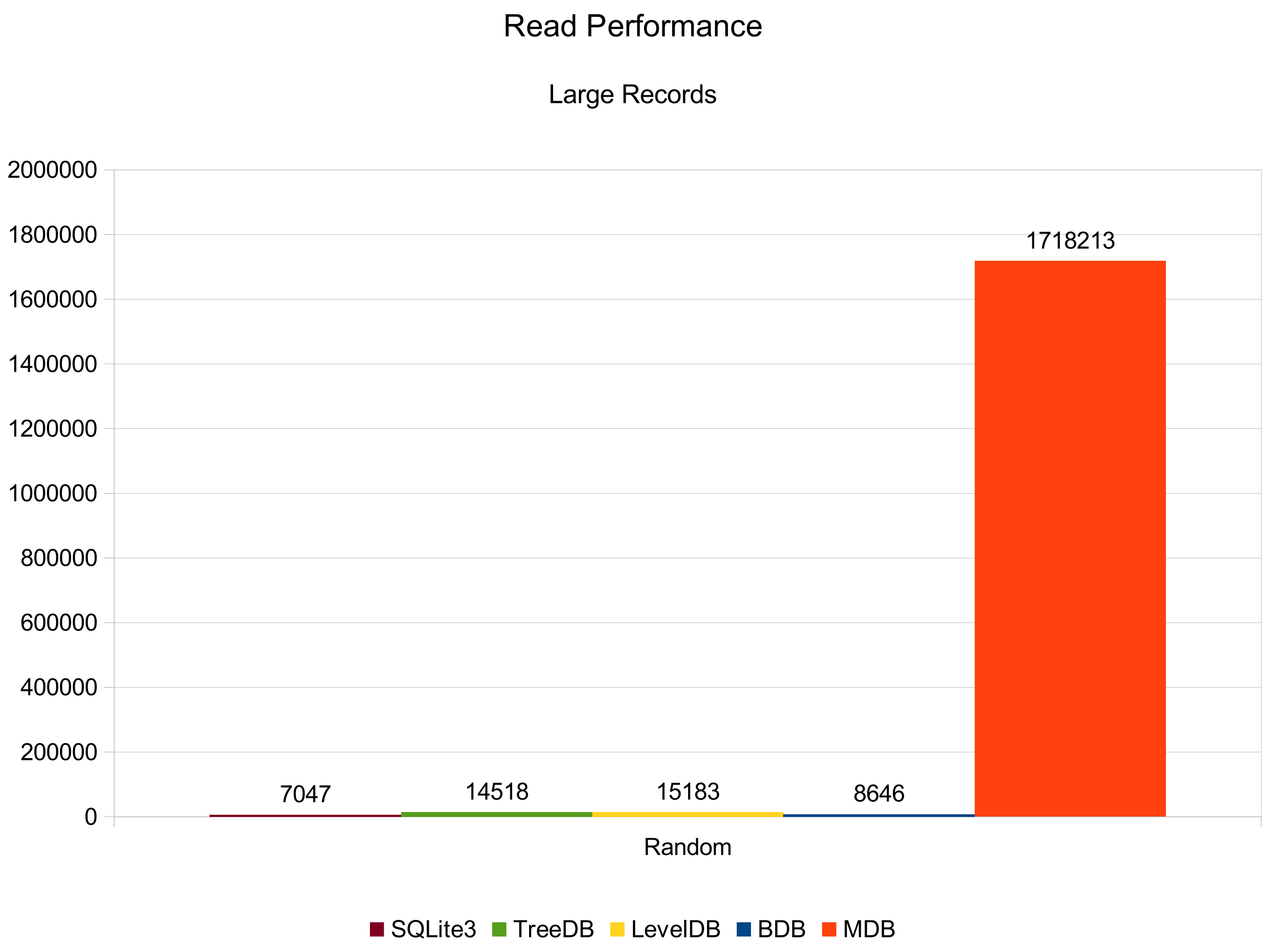
Microbenchmark Results



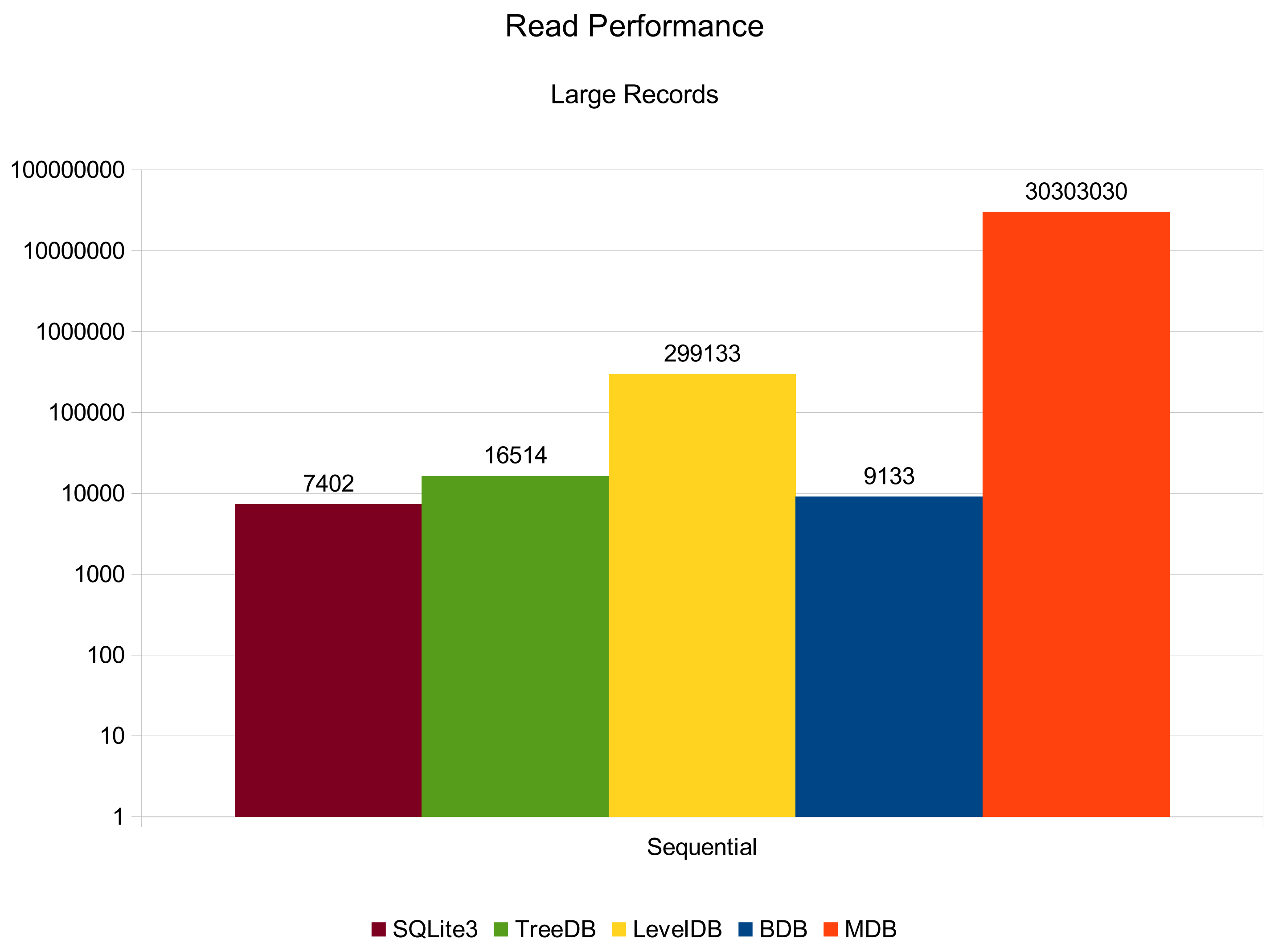


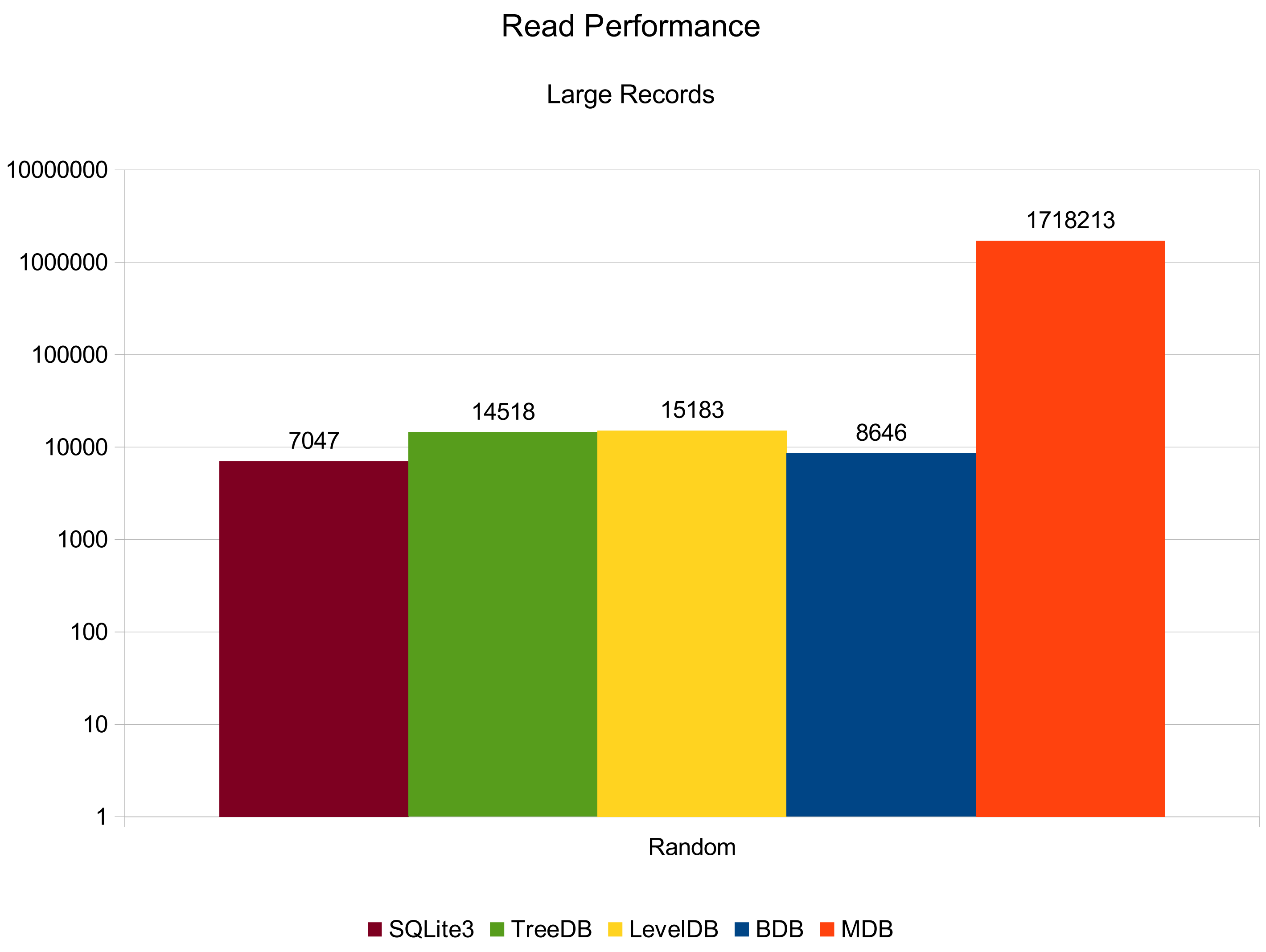
Microbenchmark Results



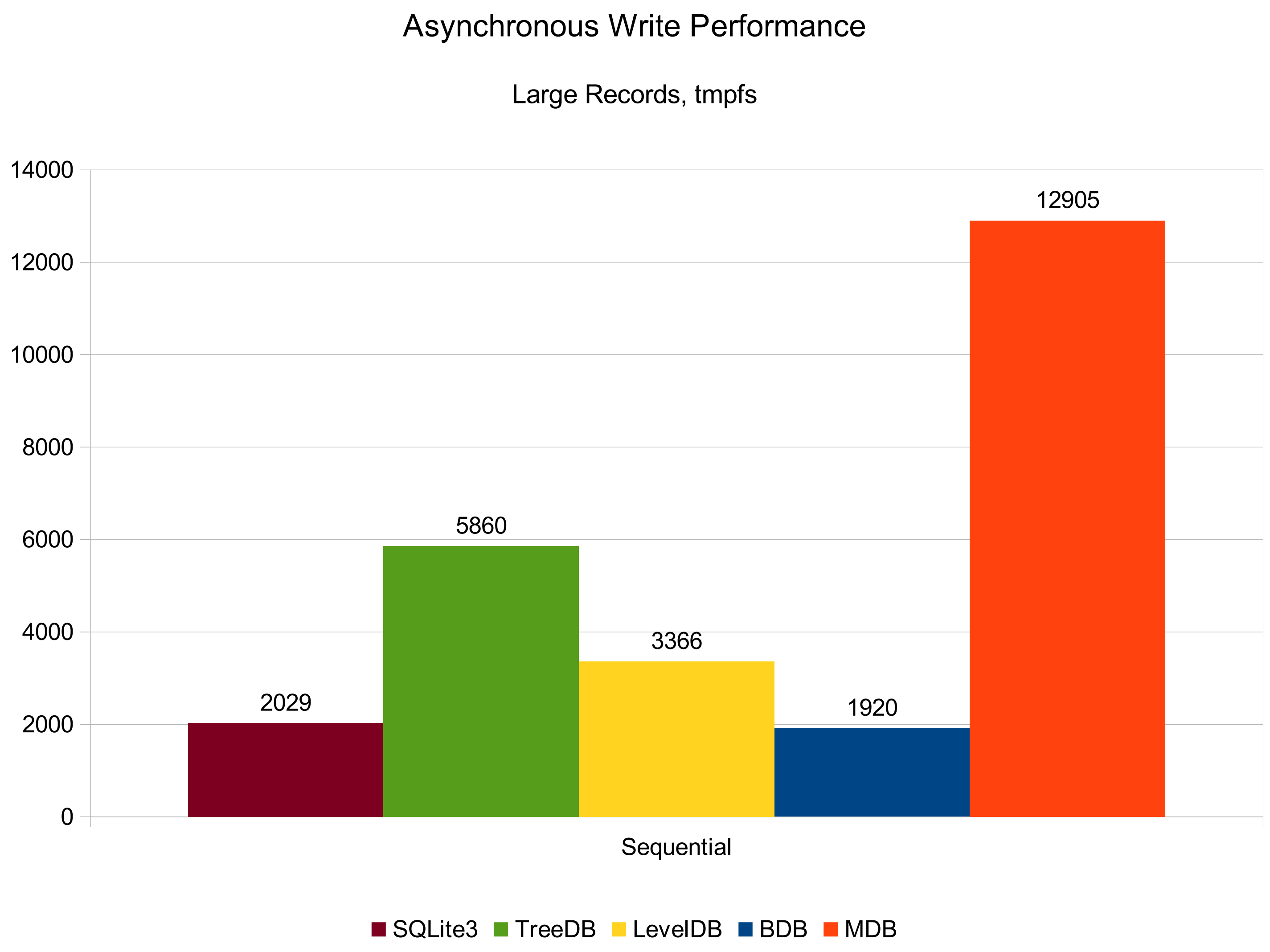


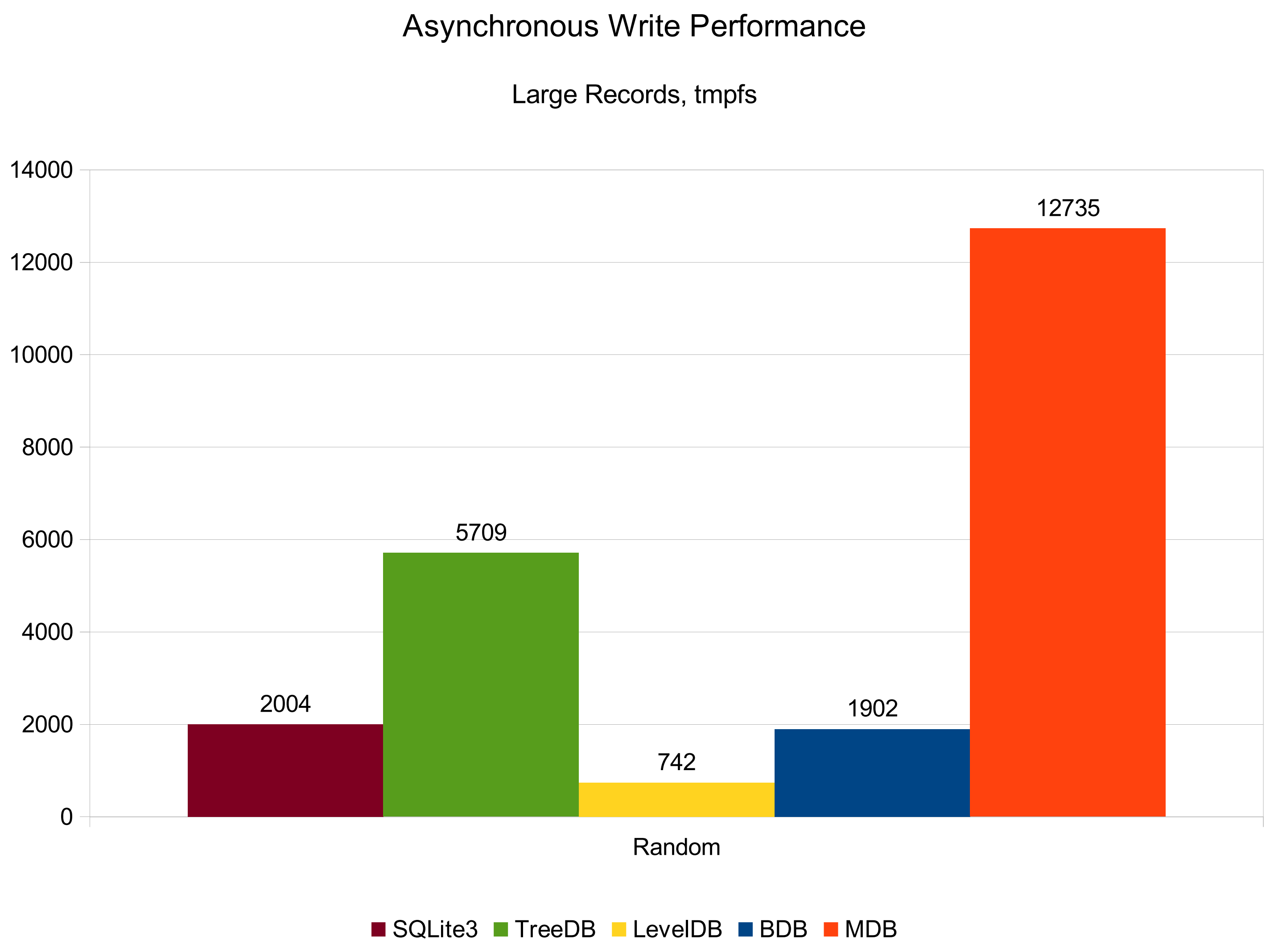
Microbenchmark Results



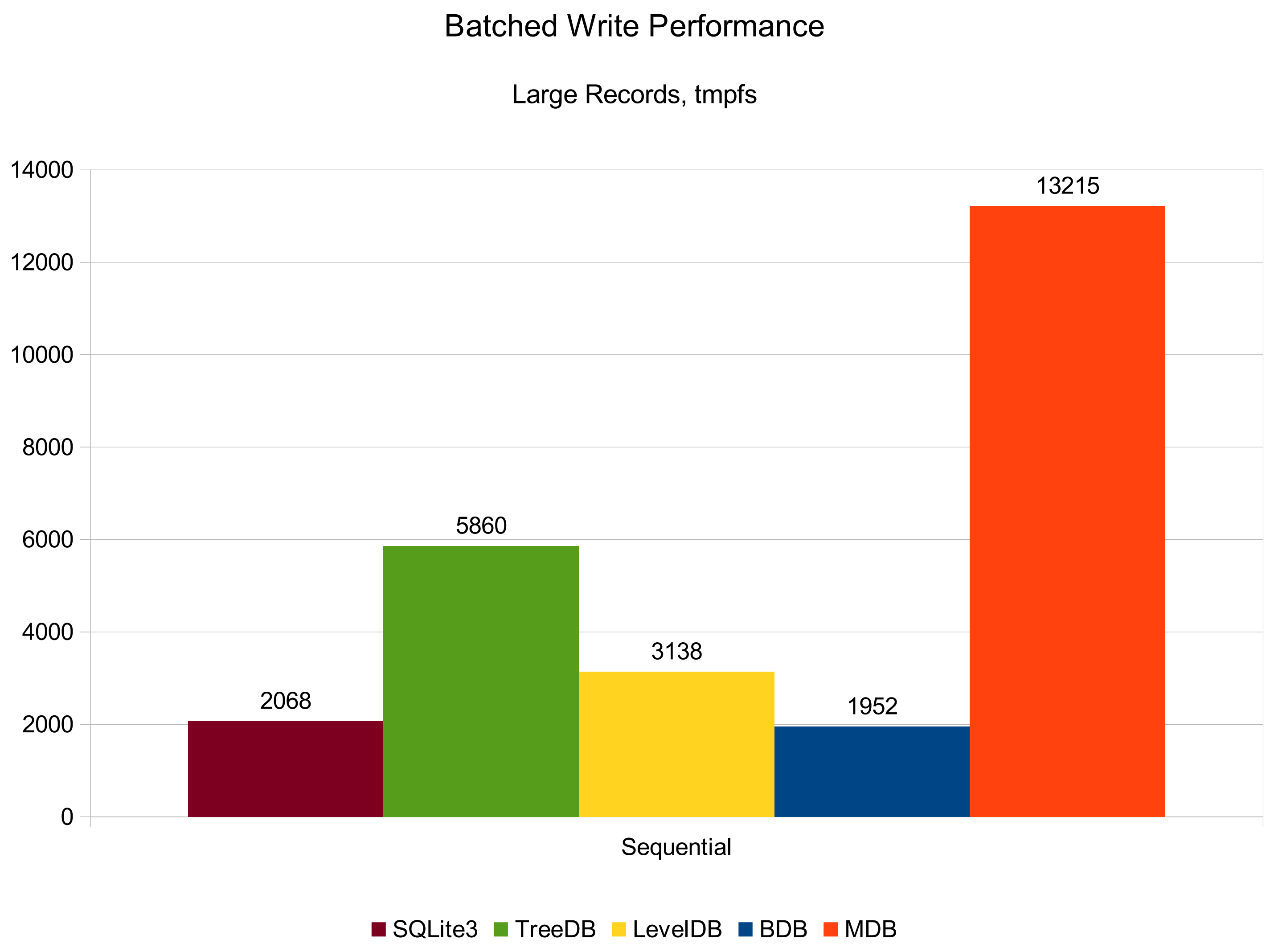


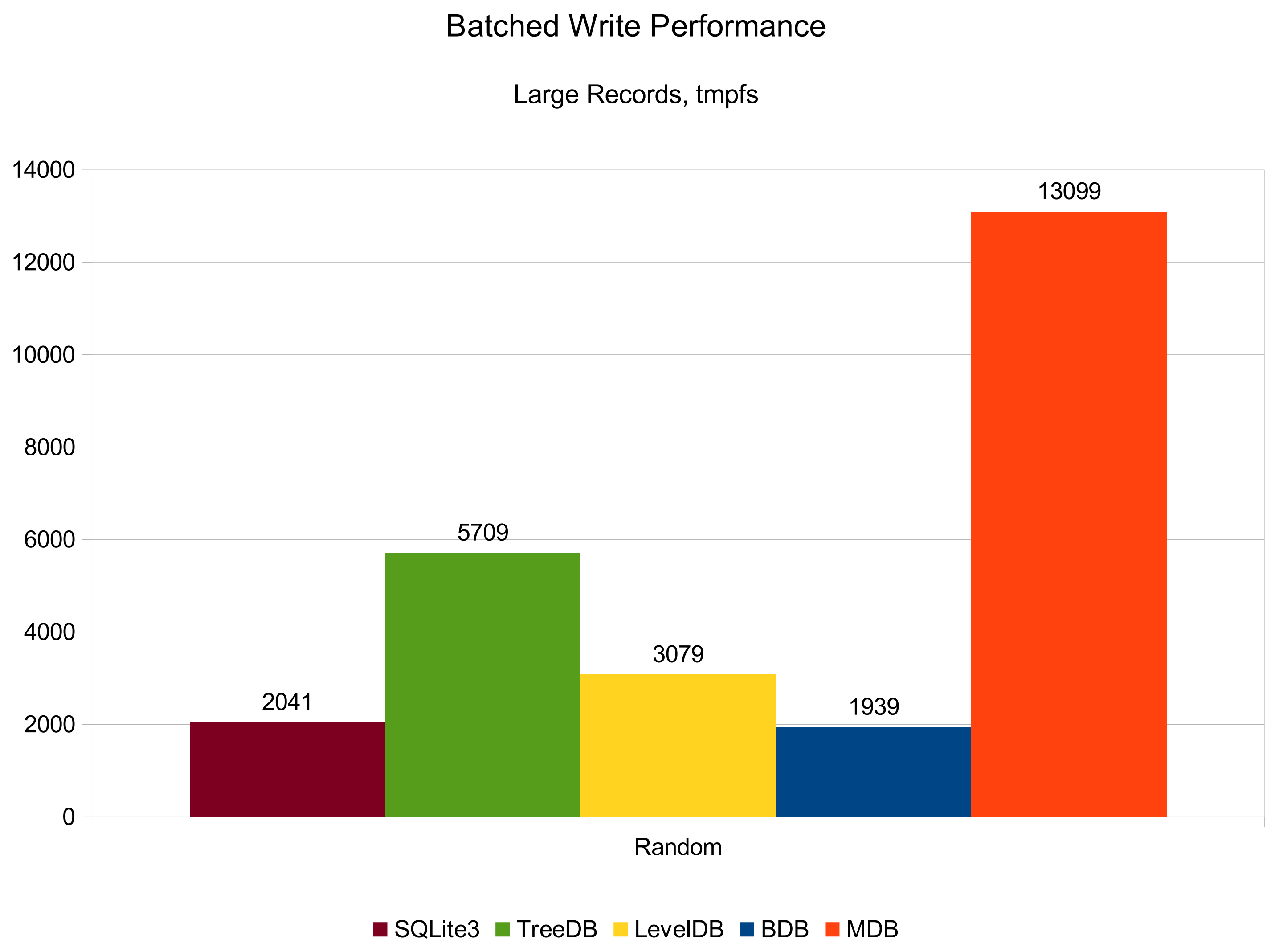
Microbenchmark Results



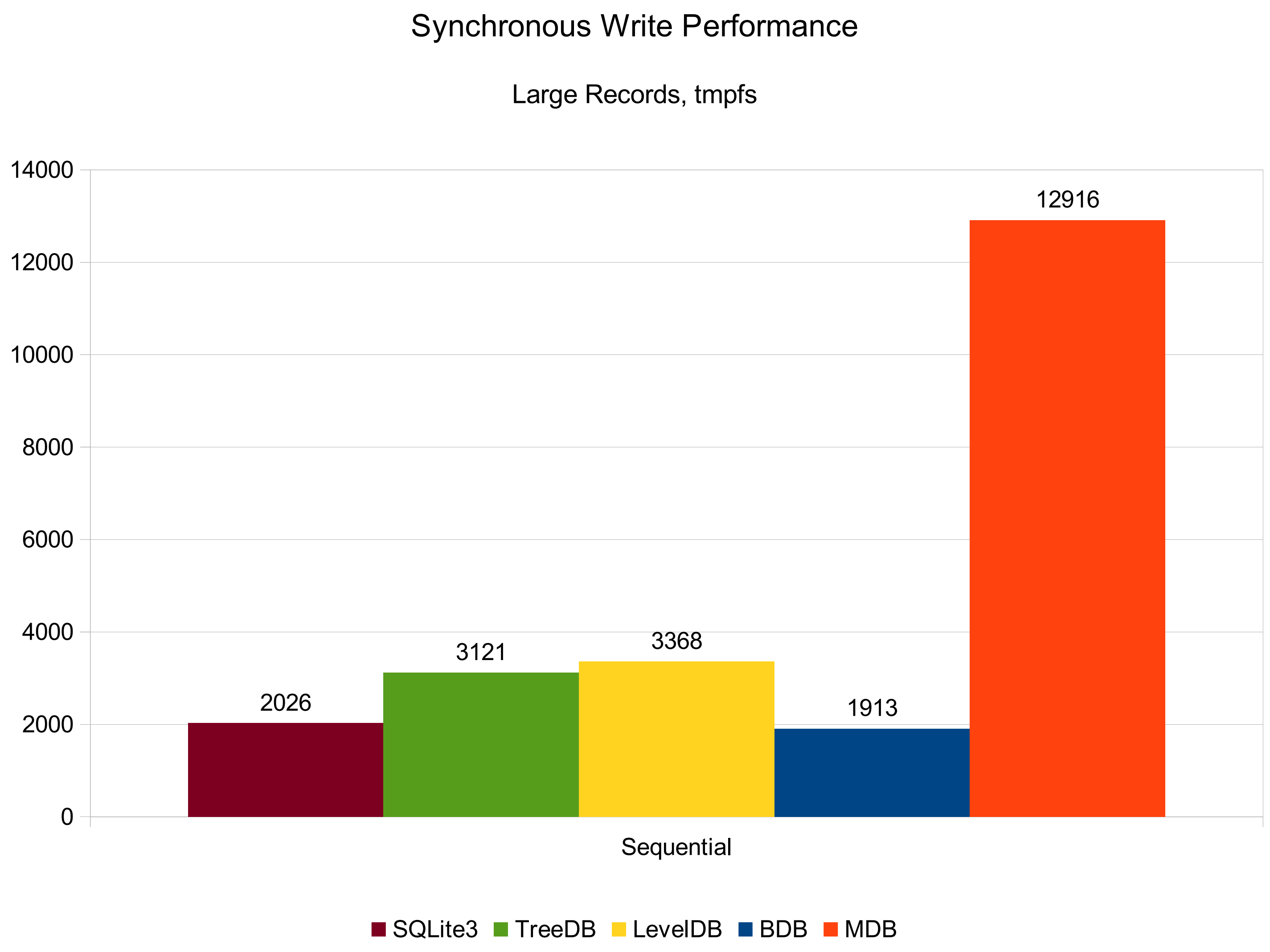


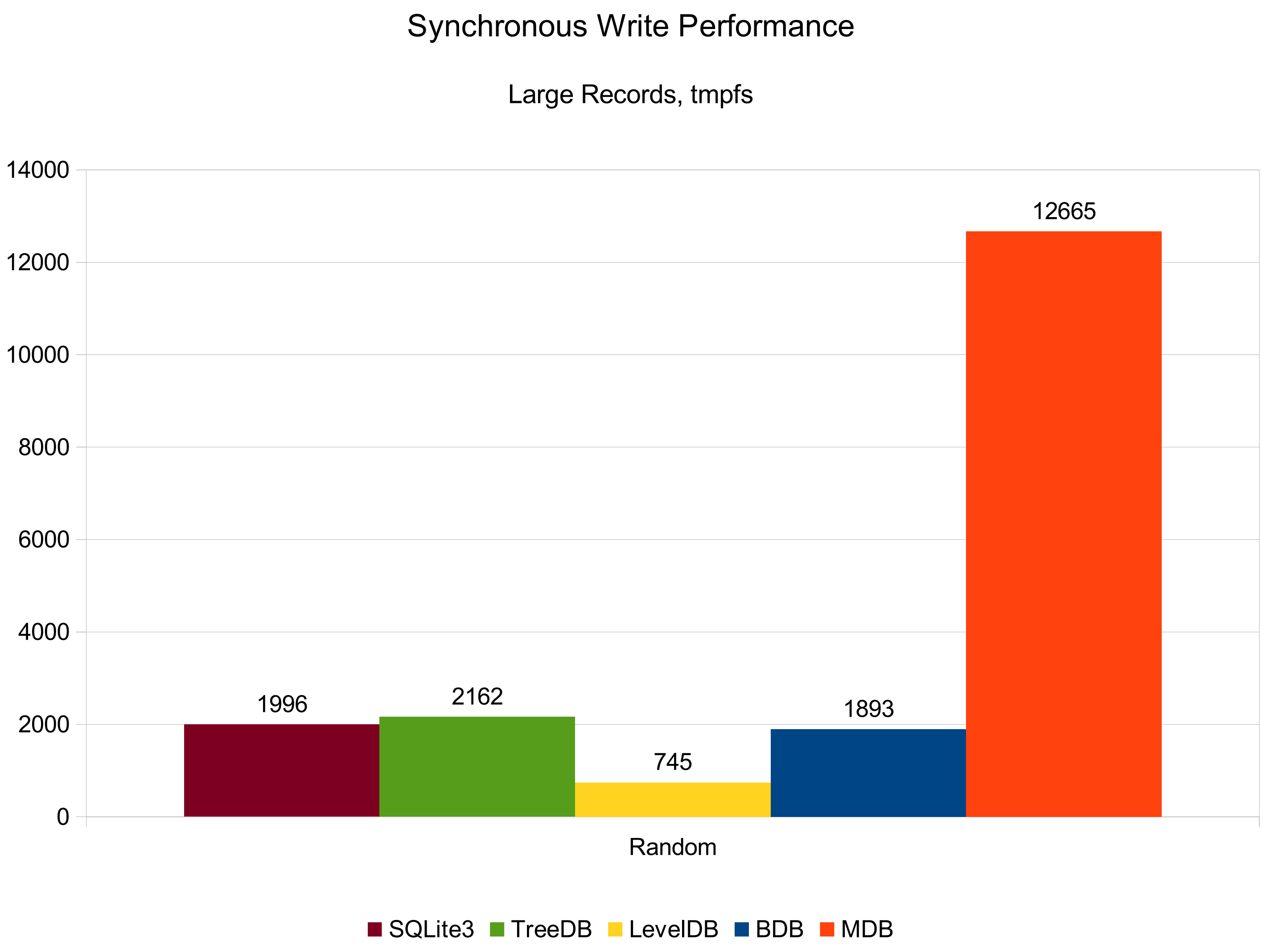
Microbenchmark Results



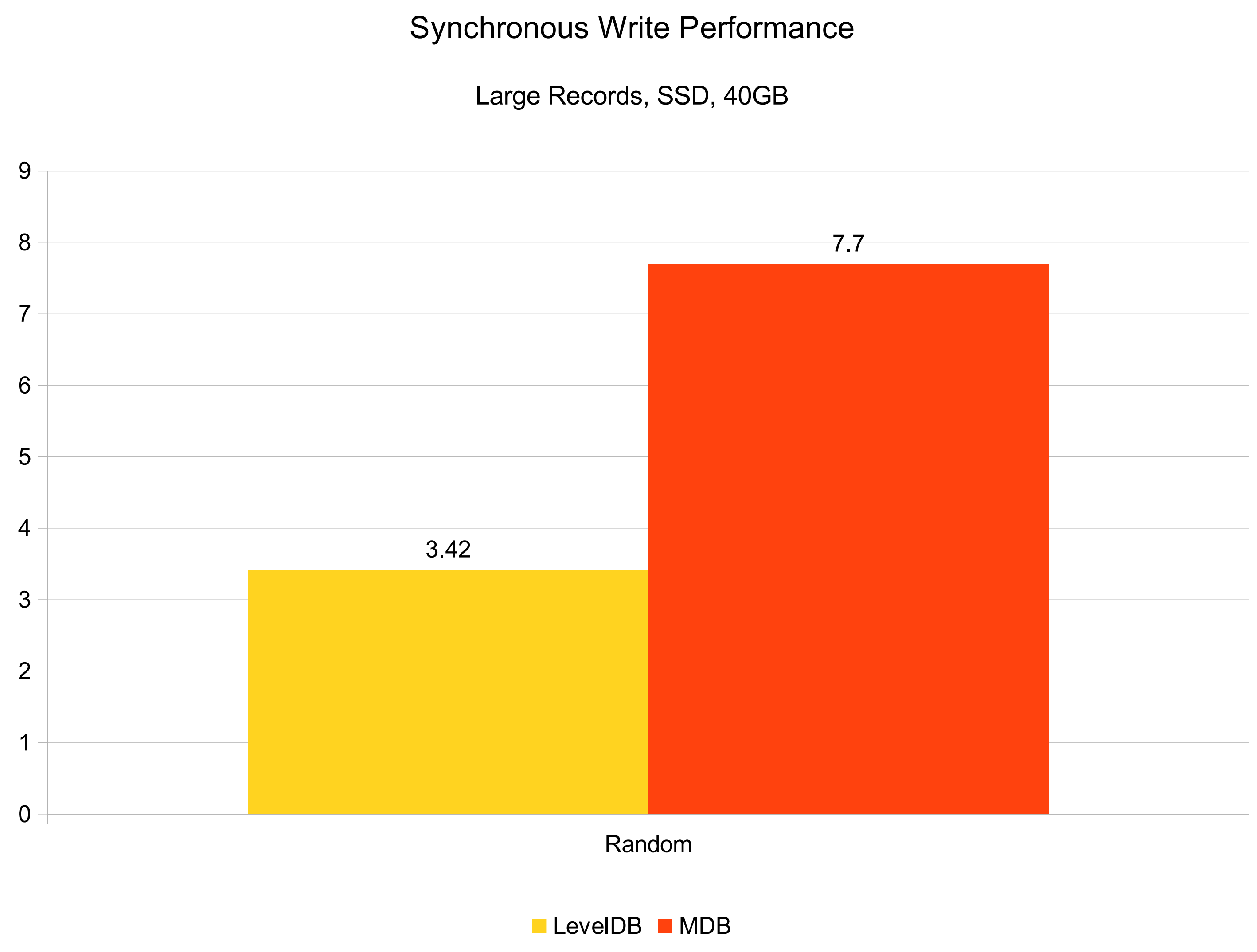


Microbenchmark Results





Microbenchmark Results



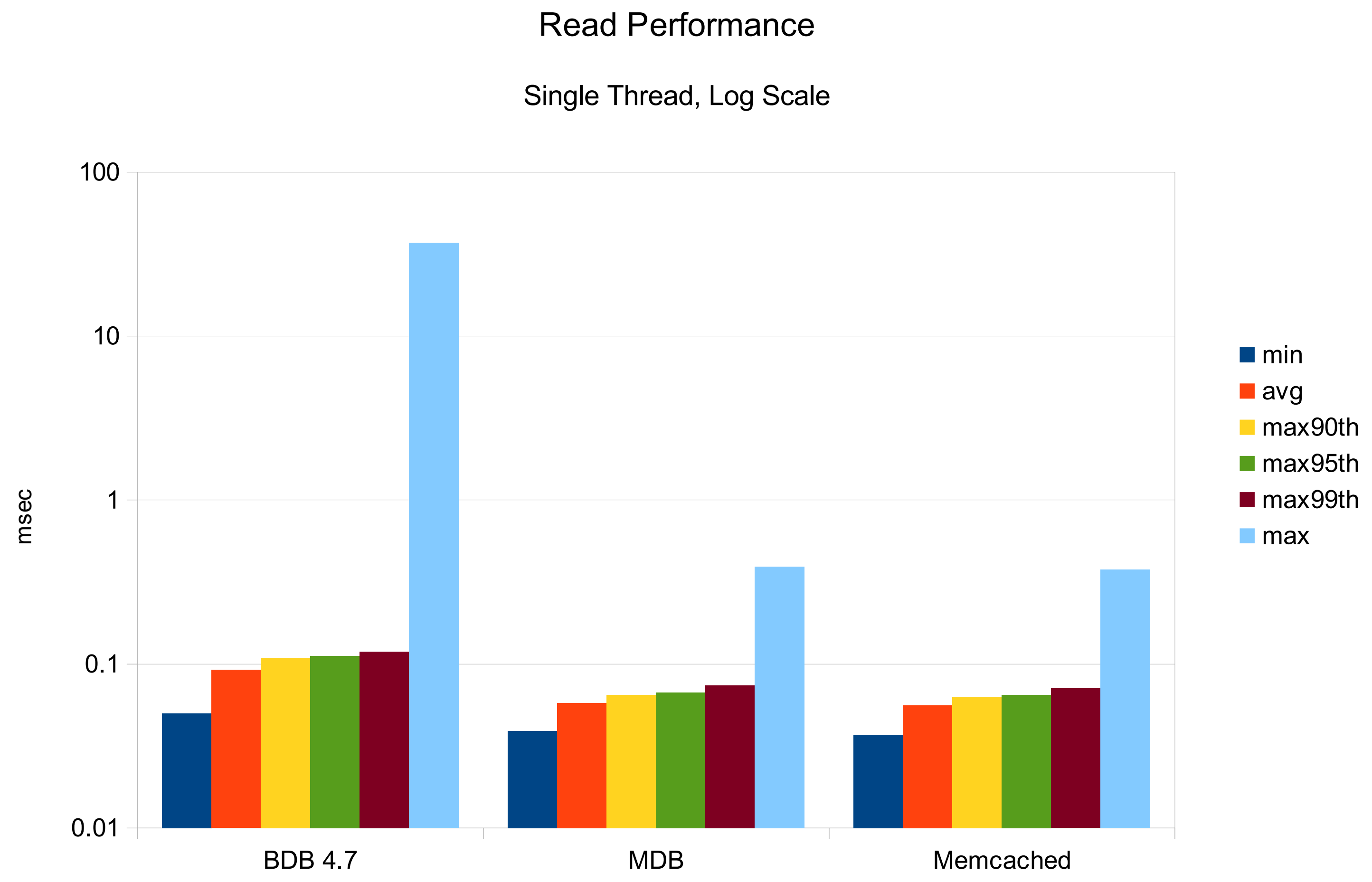
- Test random write performance when DB is 5x larger than RAM
- Supposedly a best case for LevelDB and worst case for B-trees
- Result in MB/sec, higher is better

Benchmarking…

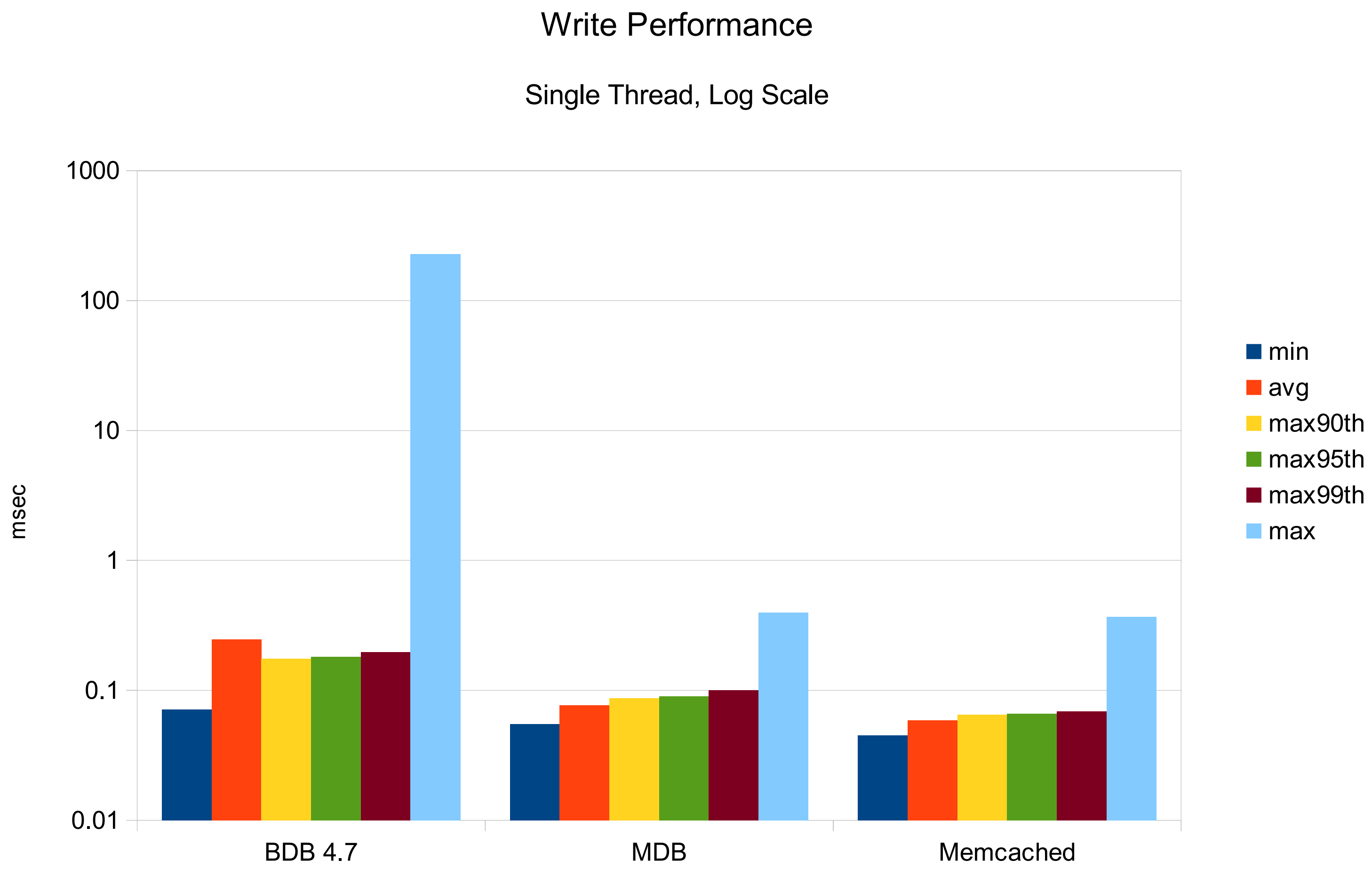
LMDB in real applications

- MemcacheDB, tested with memcachetest
- The OpenLDAP slapd server, using the back-mdb slapd backend

MemcacheDB

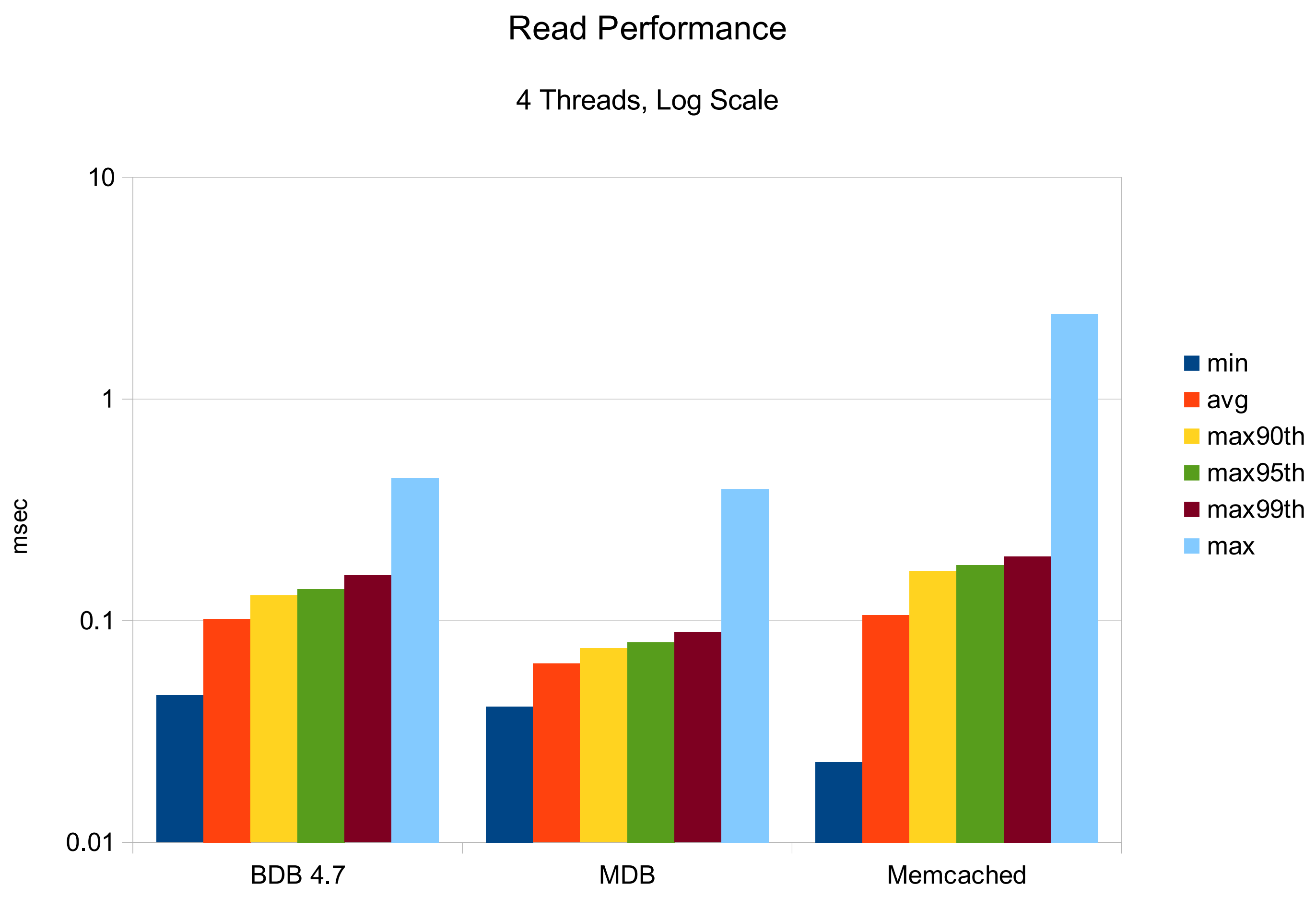


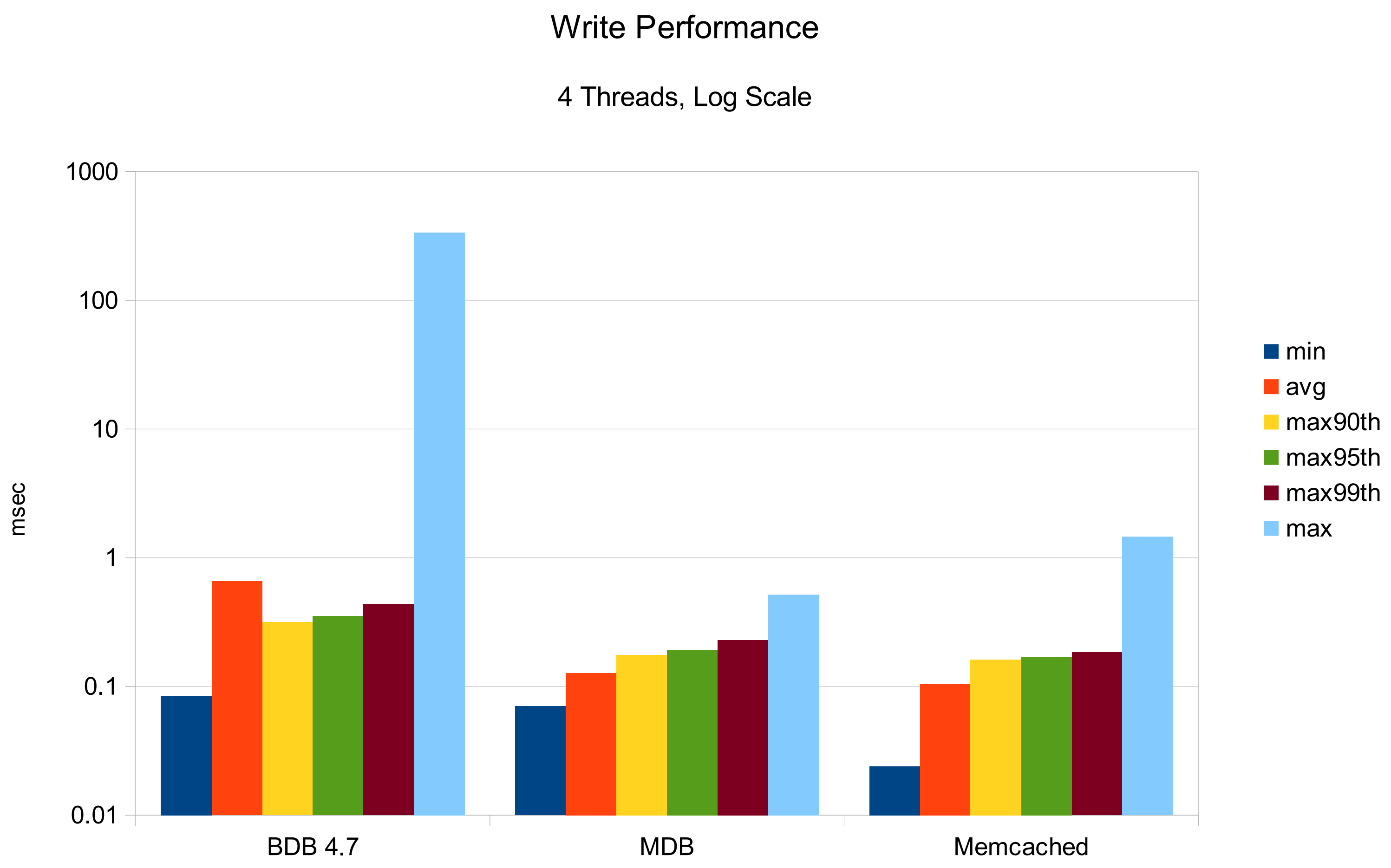
Read Performance
Single Thread, Log Scale
msec
100
10
1
0.1
0.01
BDB 4.7
MDB
Memcached
min
avg
max90th
max95th
max99th
max



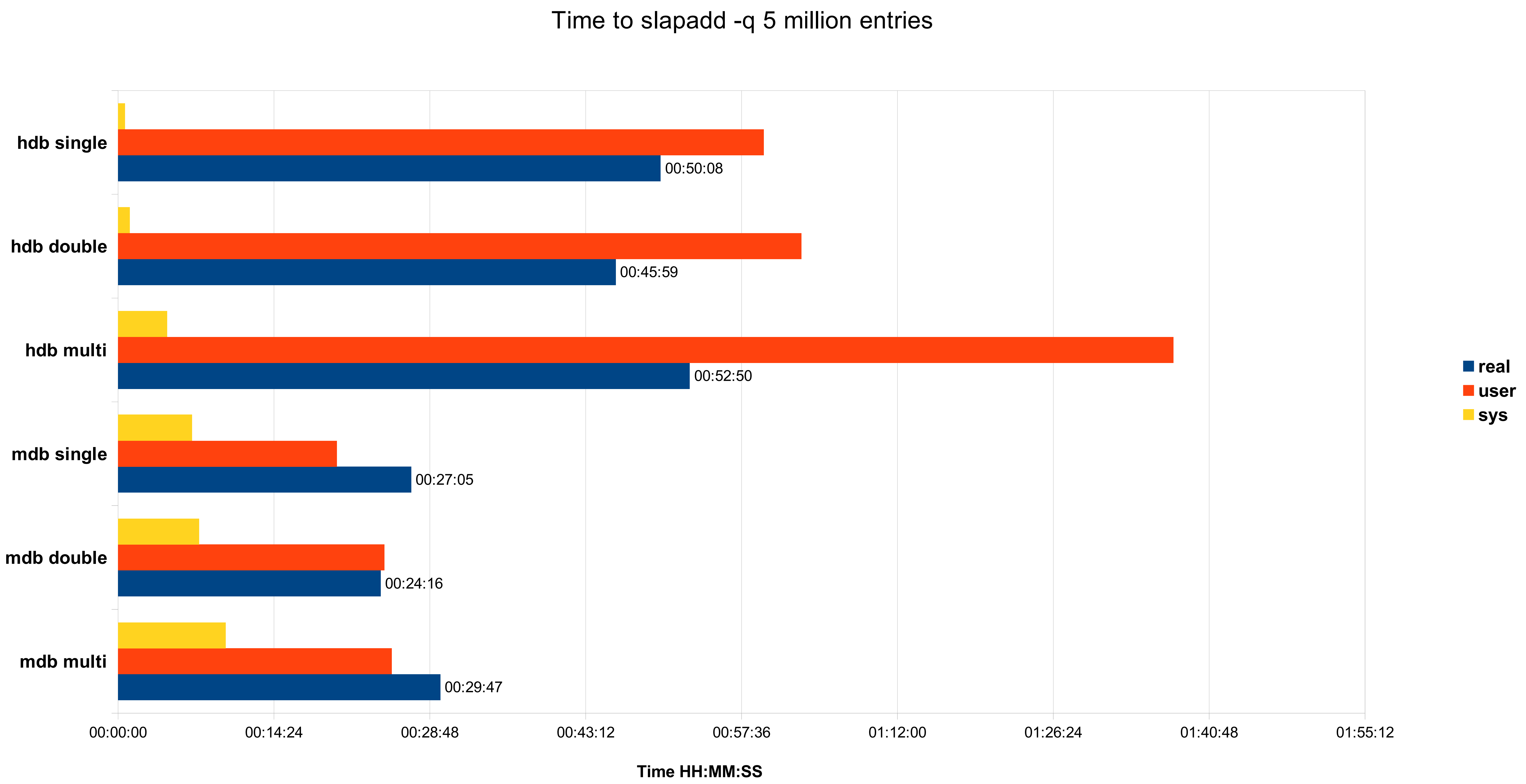
Write Performance
Single Thread, Log Scale
msec
1000
100
10
1
0.1
0.01
BDB 4.7
MDB
Memcached
min
avg
max90th
max95th
max99th
max

MemcacheDB

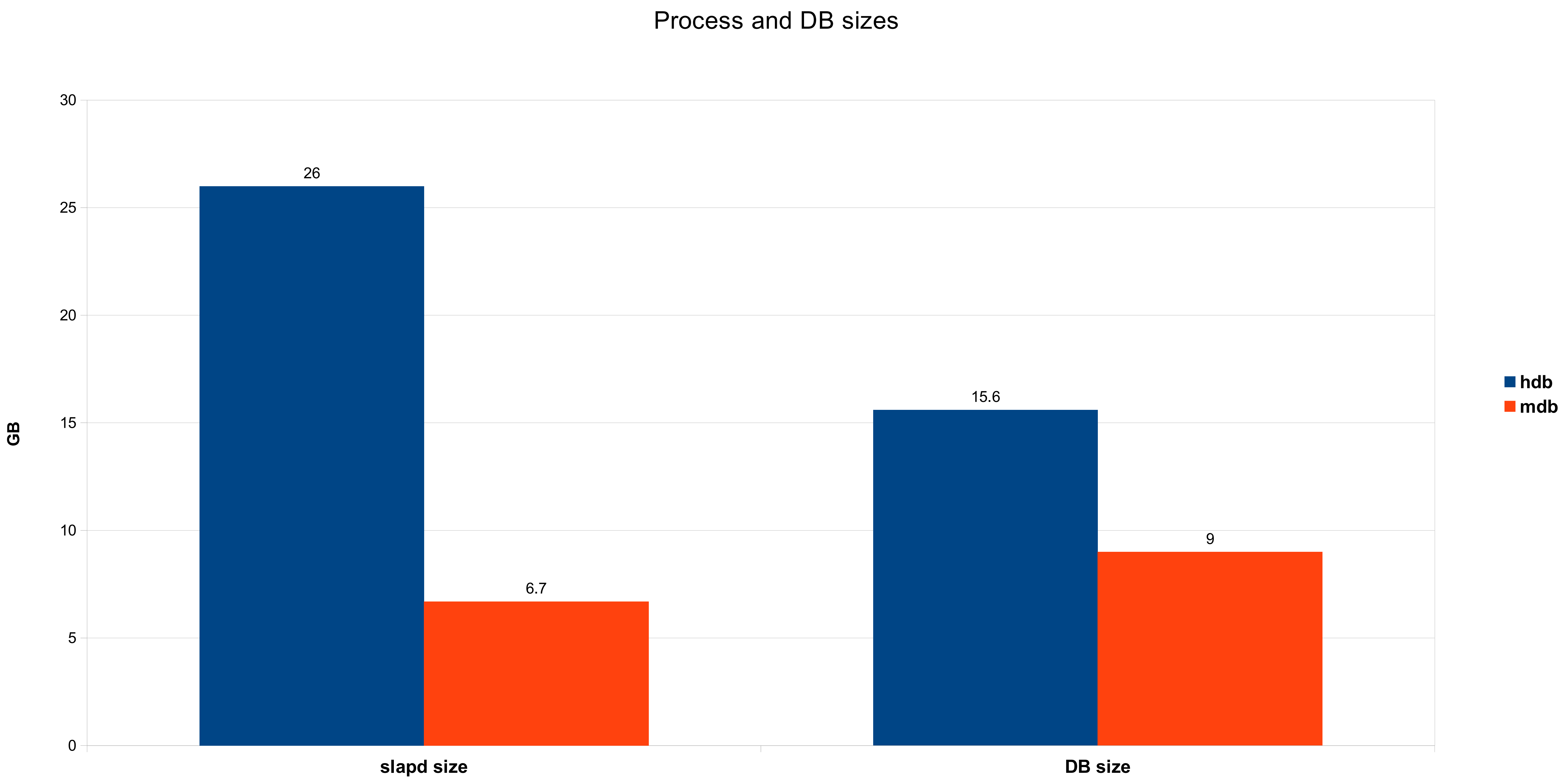

Read Performance
4 Threads, Log Scale
10
1
0.1
0.01
msec
BDB 4.7
MDB
Memcached
min
avg
max90th
max95th
max99th
max


Write Performance
4 Threads, Log Scale
1000
100
10
1
0.1
0.01
msec
BDB 4.7
MDB
Memcached
min
avg
max90th
max95th
max99th
max

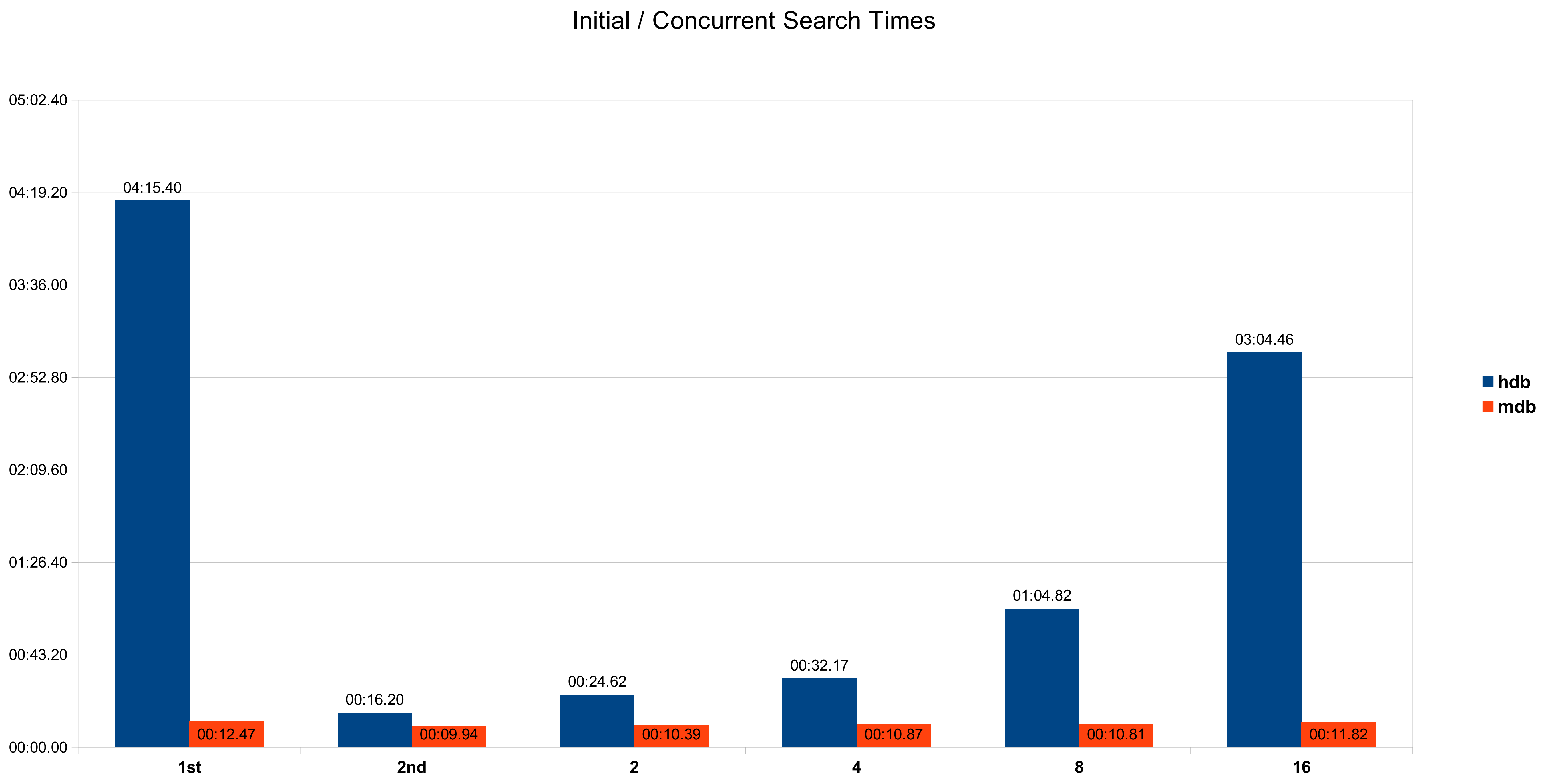
Slapd Results



Slapd Results



Slapd Results



Slapd Results